Design Patterns/Descriptions

Creational patterns, Structural patterns
and Behavioral patterns

What is a Design Pattern?

- **Origin of Pattern:** "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice." Christopher Alexander,A Pattern Language, 1977.
- **Design patterns capture the *best practices of experienced object-oriented software developers*.**
- A design patterns are **well-proved solution** for solving the specific problem/task.

Cont.…

- **Remember one-thing**,
  - design patterns are **programming language independent** strategies for solving the common OO design problems.
  - That means, a design pattern **represents an idea**, not a **particular implementation**.
  - They are reusable in multiple projects.
  - They provide the solutions that help to define the system architecture.
  - Evaluating design alternatives by:
    - –understanding costs
    - –understanding benefits of applying the pattern

Types of Design Patterns

- **Design Pattern Space**

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory method	Adapter (class)	Interpreter Template method
	Object	Abstract factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Façade Flyweight Proxy	Chain of responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Creational Patterns

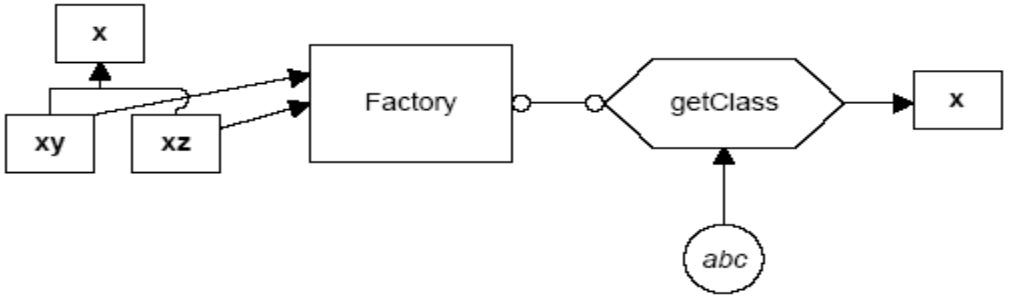
- The creational patterns deal with the best way to **create an instances of classes**.
- In Java, the simplest way to create an instance of class is by using the **new** operator.

Student = new Student(); //instance of Student class

- In many cases, the exact nature of the object that is created could vary with the needs of the program and
- Abstracting the creation process into a special “creator” class can make your program more flexible and general.

Types of creational pattern: The Factory Pattern

- The Factory pattern returns an instance of one of several possible classes depending on the data provided to it.



- Here, **x** is a base class and classes **xy** and **xz** are derived from it.
- The **Factory** is a class that decides which of these subclasses to return depending on the arguments you give it.
- The ***getClass()*** method passes in some value *abc,* and returns some instance of the class **x**.
- Which one it returns doesn't matter to the programmer since they all have the same methods, but different implementations.

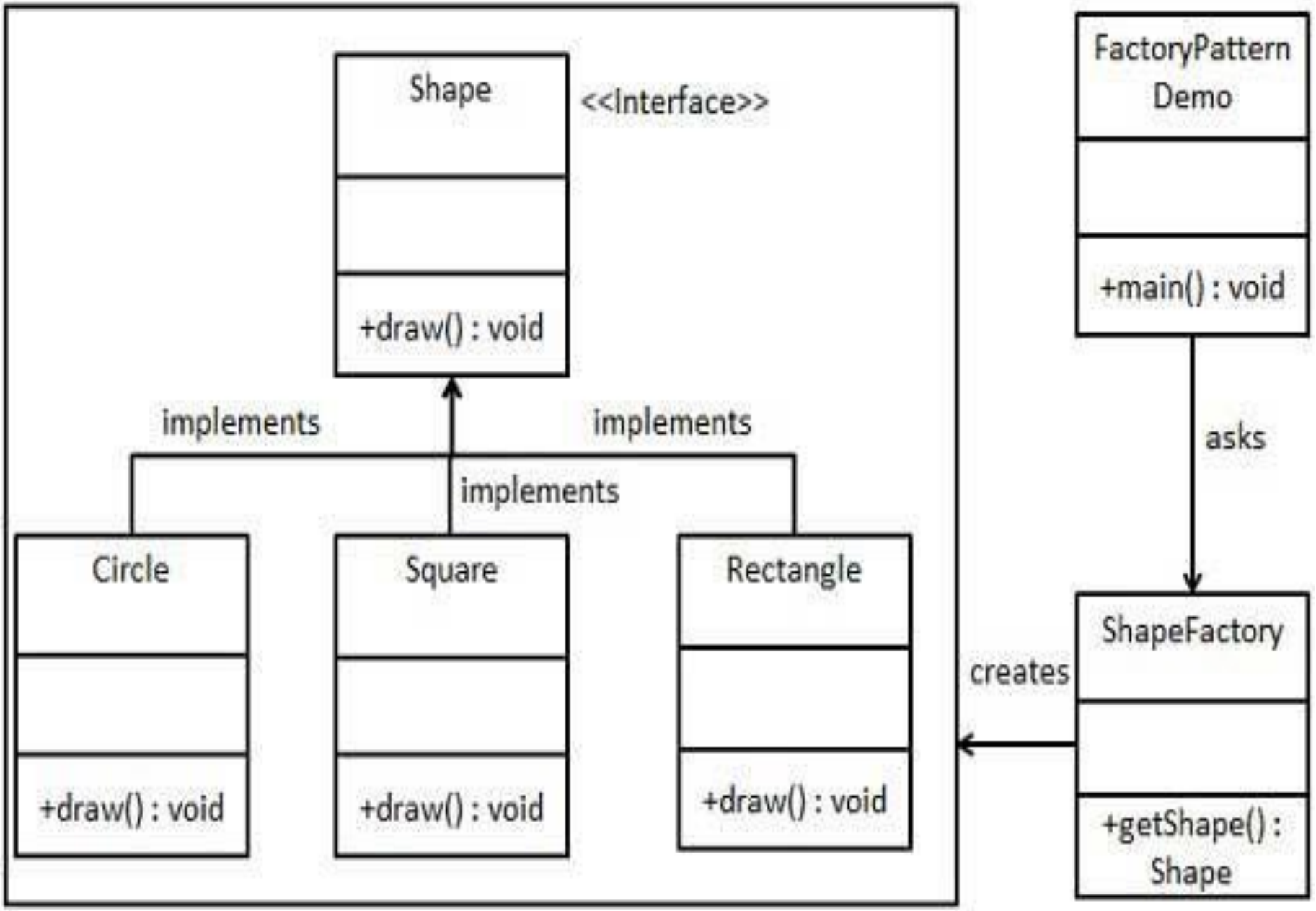
... Cont'd

```
public interface Shape {
void draw();
}

public class Circle implements Shape {
public void draw() {
System.out.println("Inside Circle::draw() method.");
}
}

public class Rectangle implements Shape {
public void draw() {
System.out.println("Inside Rectangle::draw() method.");
}
}

public class Square implements Shape {
public void draw() {
System.out.println("Inside Square::draw() method.");
}
}
```

... Cont'd

```
public class ShapeFactory {
//use getShape method to get object of type shape

public Shape getShape(String shapeType){
        if(shapeType == null){
        return null;
        }
        if(shapeType.equalsIgnoreCase("CIRCLE")){
        return new Circle();
        } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
        return new Rectangle();
        } else if(shapeType.equalsIgnoreCase("SQUARE")){
        return new Square();
        }
        return null;
}

}
```

... Cont'd


Shape
<<Interface>>
+draw() : void
implements
implements
implements
Circle
+draw() : void
Square
+draw() : void
Rectangle
+draw() : void
FactoryPattern Demo
+main() : void
asks
creates
ShapeFactory
+getShape() : Shape

… Cont’d

```
public class Client {
public static void main(String[] args) {
ShapeFactory shapeFactory = new ShapeFactory();
//get an object of Circle and call its draw method.
Shape shape1 = shapeFactory.getShape("CIRCLE");
shape1.draw();
//get an object of Rectangle and call its draw method.
Shape shape2 = shapeFactory.getShape("RECTANGLE");
//call draw method of Rectangle
shape2.draw();
//get an object of Square and call its draw method.
Shape shape3 = shapeFactory.getShape("SQUARE");
//call draw method of circle
shape3.draw();
}
}
```

When to Use a Factory Pattern?

- You should consider using a Factory pattern when:
  - A class uses its subclasses to specify which objects it creates.
  - You want to localize the knowledge of which class gets created.
- There are several similar variations on the factory pattern to recognize:
  - The base class is abstract and the pattern must return a complete working class.
  - The base class contains default methods and is only subclassed for cases where the default methods are insufficient.
  - Parameters are passed to the factory telling it which of several class types to return.
  - In this case the classes may share the same method names but may do something quite different.

The Abstract Factory Pattern

- The Abstract Factory pattern is one level of abstraction higher than the factory pattern.
- In other words, **the Abstract Factory is a factory object that returns one of several factories.**

… Cont’d

```
public interface Color {
void fill();
}

public class Blue implements Color {

public void fill() {
System.out.println("Inside Blue::fill() method.");
}
}

public class Red implements Color {
public void fill() {
System.out.println("Inside Red::fill() method.");
}}

public class Green implements Color {
public void fill() {
System.out.println("Inside Green::fill() method.");
}
}
```

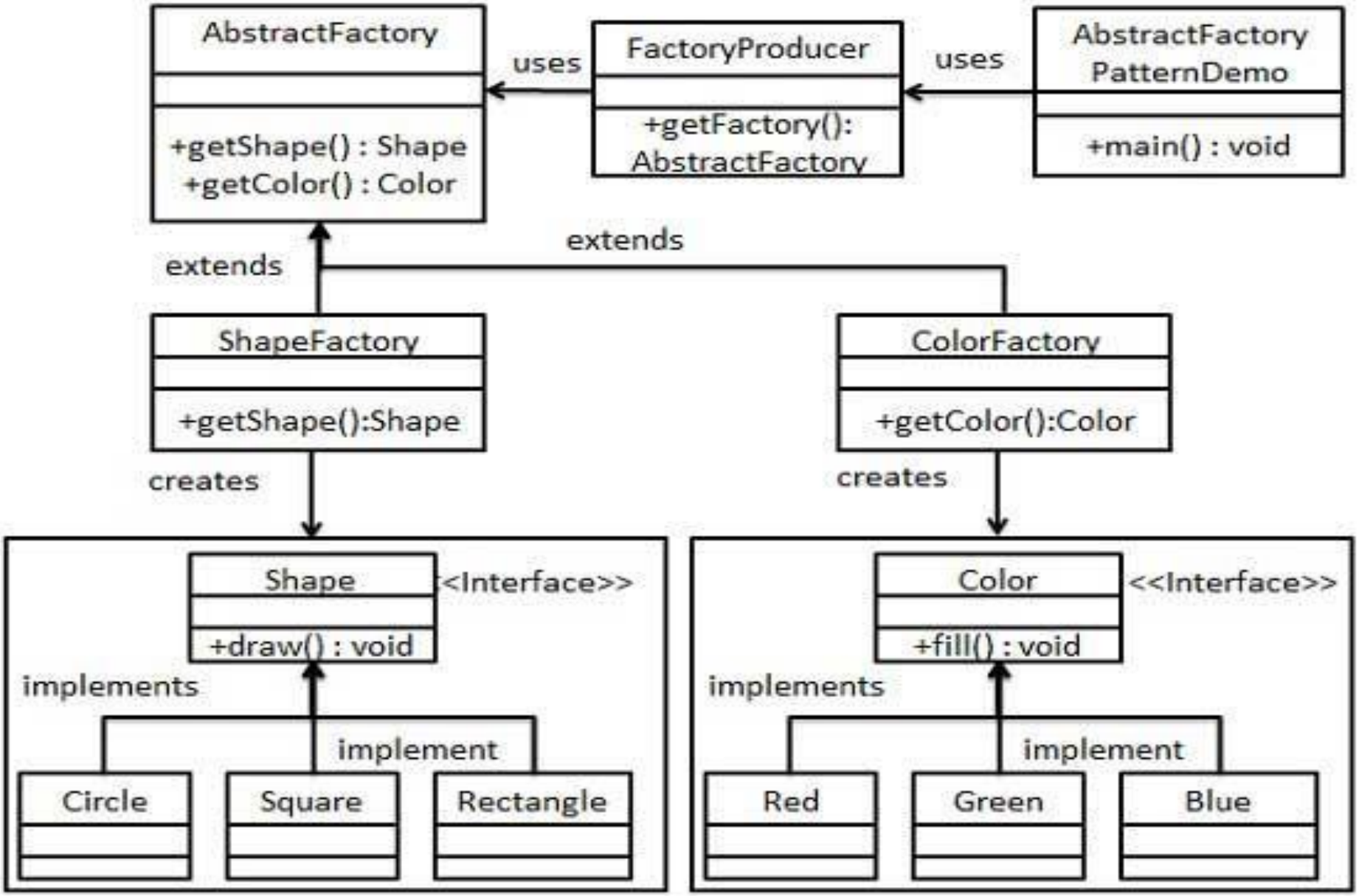
...Cont'd

```
public abstract class AbstractFactory {
abstract Color getColor(String color);
abstract Shape getShape(String shape) ;
}

public class ColorFactory extends AbstractFactory {
            public Shape getShape(String shapeType){
            return null;
            }

      Color getColor(String color) {    if(color == null){
                return null;    }
            if(color.equalsIgnoreCase("RED")){    return new Red();
            } else if(color.equalsIgnoreCase("GREEN")){
            return new Green();
            } else if(color.equalsIgnoreCase("BLUE")){
            return new Blue();   }
            return null;
}
}
```

... Cont’d

```
public class FactoryProducer {
public static AbstractFactory getFactory(String choice){
if(choice.equalsIgnoreCase("SHAPE")){
return new ShapeFactory();
} else if(choice.equalsIgnoreCase("COLOR")){
return new ColorFactory();
}
return null;
}
}
```

... Cont'd


AbstractFactory
+getShape() : Shape
+getColor() : Color
uses
FactoryProducer
+getFactory(): AbstractFactory
uses
AbstractFactory PatternDemo
+main() : void
extends
extends
ShapeFactory
+getShape():Shape
ColorFactory
+getColor():Color
creates
creates
Shape
<<Interface>>
+draw() : void
implements
implement
Circle
Square
Rectangle
Color
<<Interface>>
+fill() : void
implements
implement
Red
Green
Blue

... Cont'd

```
public class Client {
public static void main(String[] args) {          //get shape factory
AbstractFactory shapeFactory = FactoryProducer.getFactory("SHAPE");
//get an object of Shape Circle
Shape shape1 = shapeFactory.getShape("CIRCLE");
//call draw method of Shape Circle
shape1.draw();       //get an object of Shape Rectangle
Shape shape2 = shapeFactory.getShape("RECTANGLE");
//call draw method of Shape Rectangle
shape2.draw();      //get an object of Shape Square
AbstractFactory colorFactory = FactoryProducer.getFactory("COLOR");
//get an object of Color Red
Color color1 = colorFactory.getColor("RED");     //call fill method of Red
color1.fill();              //get an object of Color Green
//similarly you can create objects of another color classes and call the method.

}
}
```

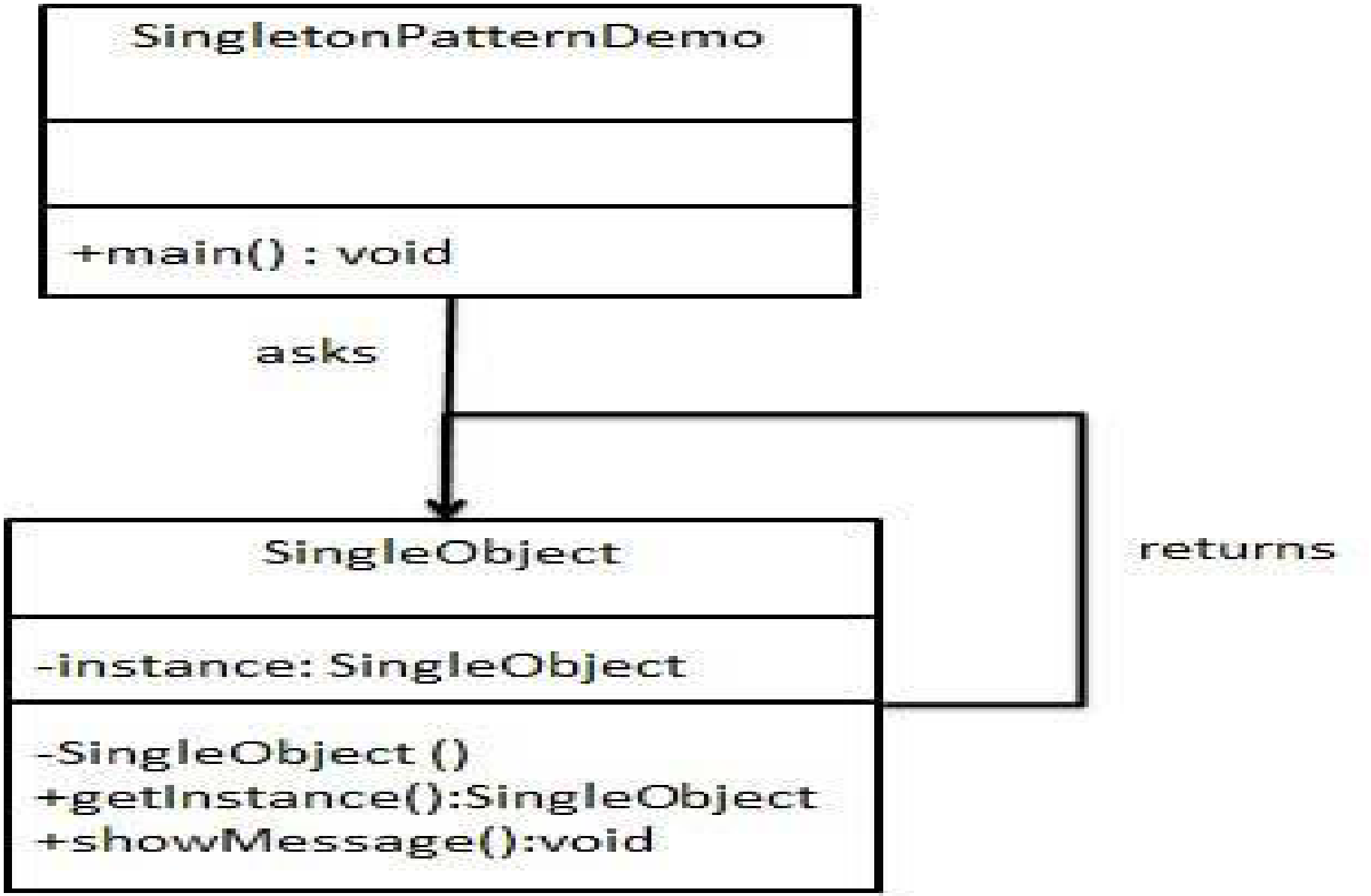
… Cont'd

- One of the main **purposes** of the **Abstract Factory** is that it **isolates the concrete classes that are generated**.
- The actual class names of these classes are hidden in **the factory** and need **not be known at the client level at all**.
- Because of the isolation of classes, you can change or interchange these product class families freely.
- While all of the classes that the Abstract Factory generates have the same base class, there is nothing to prevent some derived classes from having additional methods that differ from the methods of other classes.

Singleton Pattern

- Sometimes it i**s appropriate to have exactly one instance of a class:**
- With the Singleton design pattern you can:
  - Provide a **global point of access to the object**.
  - a class must ensure that **only single instance should be created** and **single object can be used by all other classes**.
- The class can ensure that no other instance can be created (**by intercepting requests to create new objects**), and it can provide a way to access the instance.
- Singletons maintain a static reference to the **sole singleton instance** and return a reference to that instance from a static getI*nstance()* method.

… Cont'd


SingletonPatternDemo
+main() : void
asks
SingleObject
-instance: SingleObject
-SingleObject ()
+getInstance():SingleObject
+showMessage():void
returns

… Cont'd

```
public class ClassicSingleton {
      private static ClassicSingleton instance = null;
//Early, instance will be created at load time

      private ClassicSingleton() {
         // exists only to defeat instantiation.
      }
      public static ClassicSingleton getInstance() {
            if(instance == null) {
                  instance = new ClassicSingleton();
            }
            return instance;
      }
}
```

- The *ClassicSingleton* class maintains a static reference to the alone singleton instance and returns that reference from the static getInstance() method.

… Cont'd

- The ClassicSingleton class employs a technique known as lazy instantiation to create the singleton;
  - creation of instance when required.
- As a result, the singleton instance is not created until the getInstance() method is called for the first time.
- **This technique ensures that singleton instances are created only when needed.**
- To create the singleton class, we need to have static member of class, private constructor and static factory method.
  - **Static member:** It gets memory only once because of static, it contains the instance of the Singleton class.
  - **Private constructor:** It will prevent to instantiate the Singleton class from outside the class.
  - **Static factory method:** provides the global point of access to the Singleton object and returns the instance to the caller.

… Cont'd

Advantage of Singleton design pattern

- Saves memory because object is not created at each request. Only single instance is reused again and again.
- It can be difficult to subclass a Singleton, since this can only work if the base Singleton class has not yet been instantiated.
- We can easily change a Singleton to allow a small number of instances where this is allowable and meaningful.
- We can use the same approach to control the number of instances that the application uses.
- Only the operation that grants access to the Singleton instance needs to change.

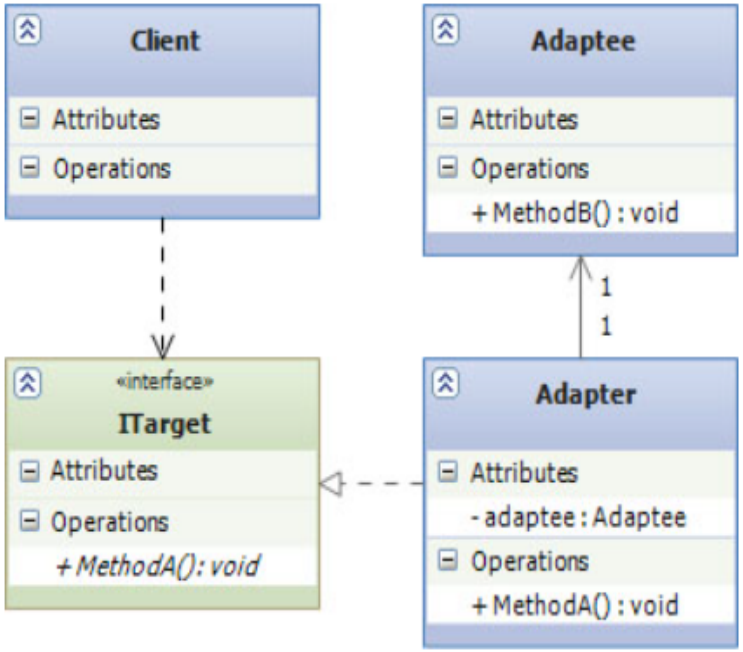
Structural Patterns

- Structural patterns describe how classes and objects can be combined to form larger structures.
- These patterns focus on, how the classes inherit from each other and how they are composed from other classes.
- The Structural patterns are:
  - **Adapter**-Match interfaces of different classes
  - **Façade**-A single class that represents an entire subsystem
- **Facade** defines a new interface, whereas **Adapter** reuses an old interface.
- Remember that **Adapter** makes two existing interfaces work together as opposed to defining an entirely new one.
- **Facade** objects are often **Singleton** because only one **Facade** object is required.
- **Abstract Factory** can be used as an alternative to **Facade** to hide platform-specific classes.

Adapter Pattern

- This type of design pattern combines the capability of two independent interfaces or incompatible interfaces.
  - A real life example could be a case of card reader which acts as an adapter between memory card and a laptop.
  - You plug-in the memory card into card reader and card reader into the laptop so that memory card can be read via laptop.
- **Usage**:
  - When you want to create a reusable class that cooperates with classes which don't have compatible interfaces.
  - When an object needs to utilize an existing class with an incompatible interface.

Structural code example



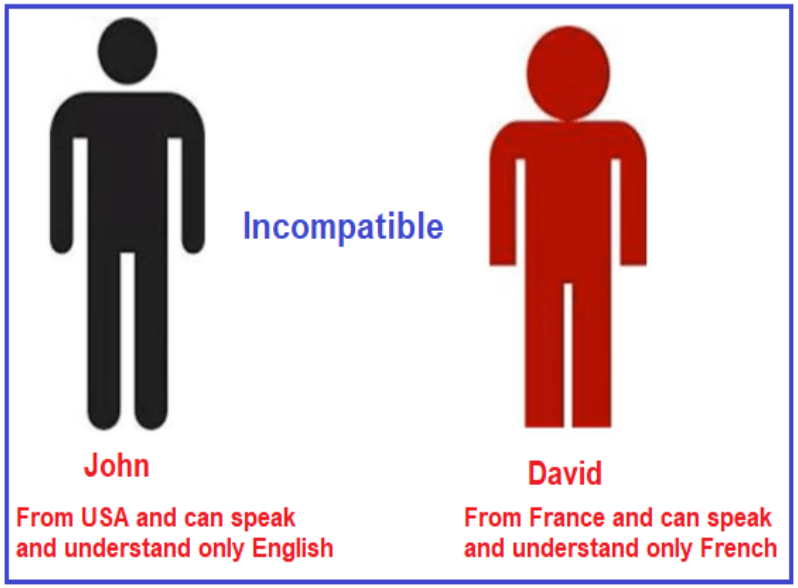
The UML diagram below describes an implementation of the adapter design pattern. Diagram consists of four parts:

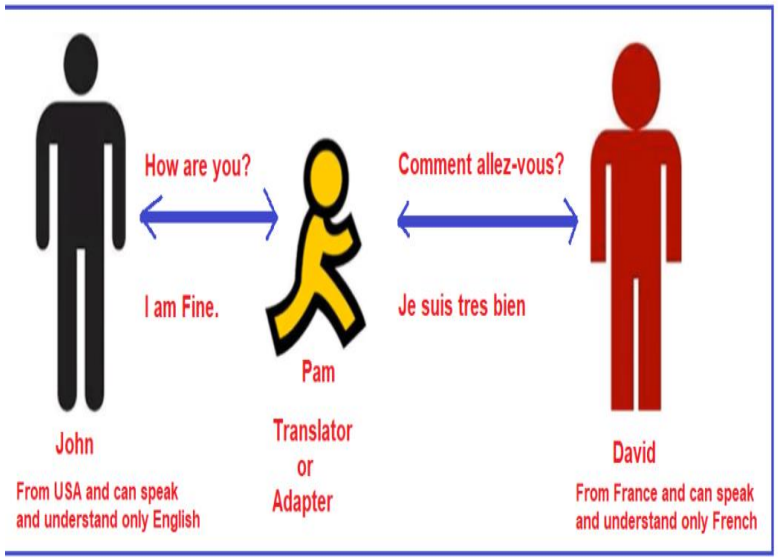
- **Client:** represents the class which need to use an incompatible interface. This incompatible interface is implemented by `Adaptee`.
- **ITarget:** defines a domain-specific interface that client uses. In this case it is an simple interface, but in some situations it could be an abstract class which adapter inherits. In this case methods of this abstract class must b overriden by concrete adapter.
- **Adaptee:** represents a class provides a functionality that is required by client.
- **Adapter:** is concrete implementation of adapter. This class translates incompatible interface of Adaptee into interface of `Client`.

Adapter Design Pattern Real-time Example- Language Translator

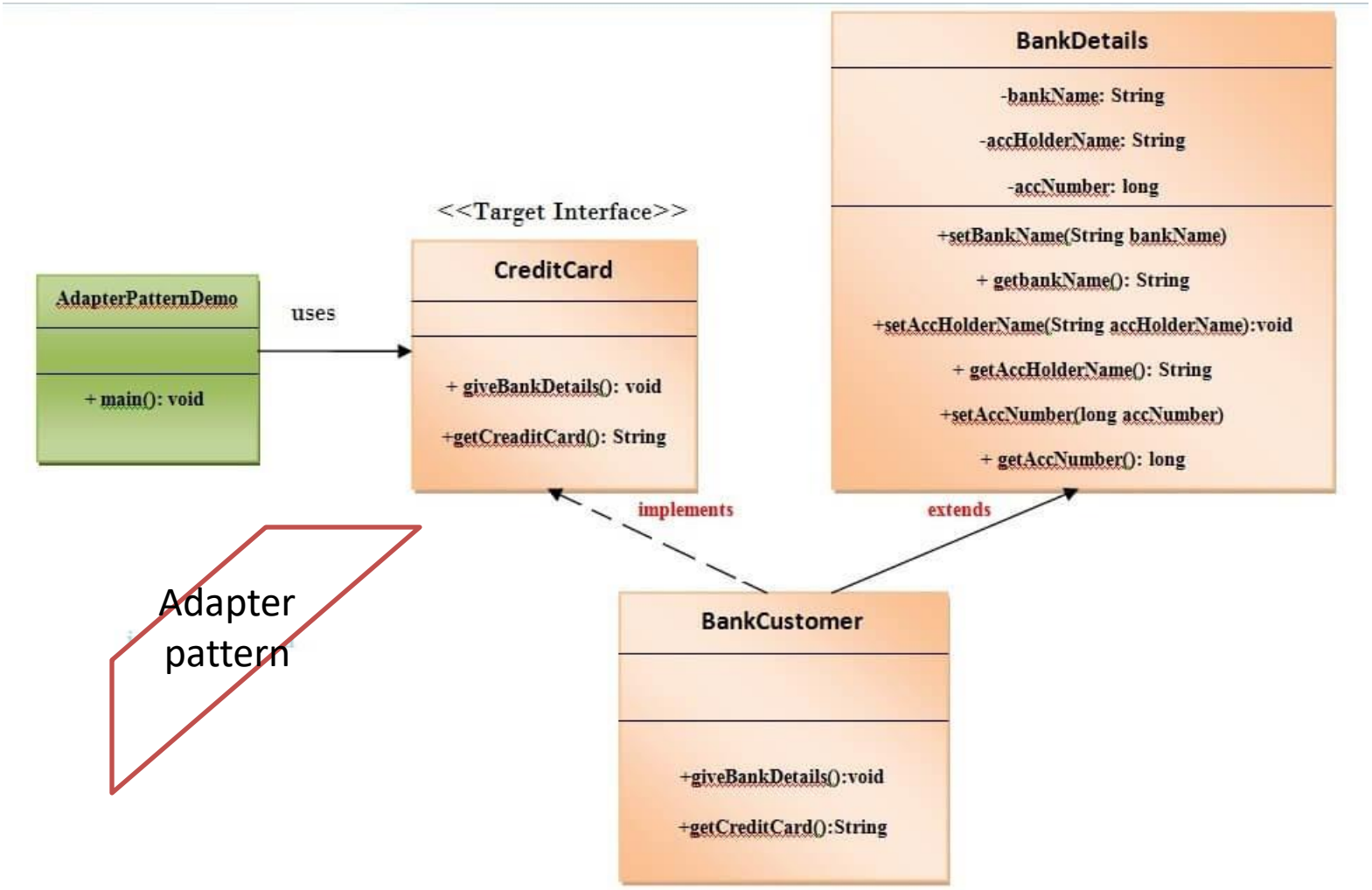
Please have a look at the following diagram.

- On the left-hand side, you can see a person called John and on the right-hand side, you can see a person called David.
- In order to make them compatible with each other, what we need to do is, we need to introduce a translator between John and David which is shown in the below image.





…UML Diagram for Credit Card



... Cont'd

Step 1: *Create a CreditCard interface (Target interface).*

```
public interface CreditCard {
   public void giveBankDetails();
   public String getCreditCard();
}// End of the CreditCard interface.
```

Step 2: Create a **BankDetails** class (Adaptee class).

File: BankDetails.java

```
// This is the adapter class.
public class BankDetails{
   private String bankName;
   private String accHolderName;
   private long accNumber;

   public String getBankName() {
      return bankName;
   }
   public void setBankName(String bankName) {
      this.bankName = bankName;
   }
```

…Cont'd

```
public String getAccHolderName() {
        return accHolderName;
    }
    public void setAccHolderName(String accHolderName) {
        this.accHolderName = accHolderName;
    }
    public long getAccNumber() {
        return accNumber;
    }
    public void setAccNumber(long accNumber) {
        this.accNumber = accNumber;
    }
}// End of the BankDetails class.
```

***Step 3**: Create a **BankCustomer** class (Adapter class).*

```
//File: BankCustomer.java
// This is the adapter class
import java.io.BufferedReader;
import java.io.InputStreamReader;
public class BankCustomer extends BankDetails implements CreditCard {
 public void giveBankDetails(){
  try{
   BufferedReader br=new BufferedReader(new InputStreamReader(System.in));

   System.out.print("Enter the account holder name :");
   String customername=br.readLine();
   System.out.print("\n");

   System.out.print("Enter the account number:");
   long accno=Long.parseLong(br.readLine());
   System.out.print("\n");
```

… Cont'd

```
System.out.print("Enter the bank name :");
  String bankname=br.readLine();

  setAccHolderName(customername);
  setAccNumber(accno);
  setBankName(bankname);
  }catch(Exception e){
      e.printStackTrace();  }  }
 @Override
 public String getCreditCard() {
  long accno=getAccNumber();
  String accholdername=getAccHolderName();
  String bname=getBankName();

  return ("The Account number "+accno+" of "+accholdername+" in "+bname+ "
                  bank is valid and authenticated for issuing the credit card. ");
 }  }//End of the BankCustomer class.
```

... Cont'd

***Step 4**: Create a **AdapterPatternDemo** class (client class).*

File: AdapterPatternDemo.java

```
//This is the client class.
public class AdapterPatternDemo {
 public static void main(String args[]){
  CreditCard targetInterface=new BankCustomer();
  targetInterface.giveBankDetails();
  System.out.print(targetInterface.getCreditCard());
 }
}//End of the BankCustomer class.
```

Facade Pattern

- A software design pattern commonly used with OO PL.
- Practically, **every Abstract Factory** is a type of **Facade.**
- Facade pattern Providing an interface to a set of interfaces.
- Facade pattern adds an interface to existing system to hide its complexities.
- This pattern involves **a single class which provides:**

J2EE Façade Example contd..

- For example, for a banking application, you may group the interactions related to managing an account into a single facade. The use cases Create New Account, Change Account Information, View Account information, and so on all deal with the coarse-grained entity object Account. Creating a session bean facade for each use case is not recommended. Thus, the functions required to support these related use cases could be grouped into a single Session Facade called AccountSessionFacade

Real world example

- In this example we need to start computer.
  - `Computer` class acts as facade which encapsulates other complex classes represented by:
    - `HardDrive` class,
    - `Memory` class and
    - `CPU` class.
  - Each of this classes has operations which must be performed when `Start()` method of Computer class is called.

Facade pattern Usage:

- When you want to provide simple interface to a complex sub-system.
- When several dependencies exist between clients and the implementation classes of an abstraction.

...Shape Facade Example

Step1: Create a **shape** interface.

```
public interface Shape {
void draw();}
```

Step2: *Create a* ***Rectangle*** *implementation class that will implement* ***shape*** *interface.*

```
public class Rectangle implements Shape {
public void draw() {
System.out.println("Inside Rectangle::draw() method.");   }
}
```

Step3: *Create a* ***circle*** *implementation class that will implement* ***shape*** *interface*.

```
public class Circle implements Shape {
public void draw() {
System.out.println("Inside Circle::draw() method.");  }
}
```

Step4: *Create a* ***Square*** *implementation class that will implement* ***shape*** *interface.*

```
public class Square implements Shape {
public void draw() {
System.out.println("Inside Square::draw() method.");  }
}
```

... Cont'd

***Step 5**: Create a* ShapeMaker *concrete class that will use **shape** interface.*

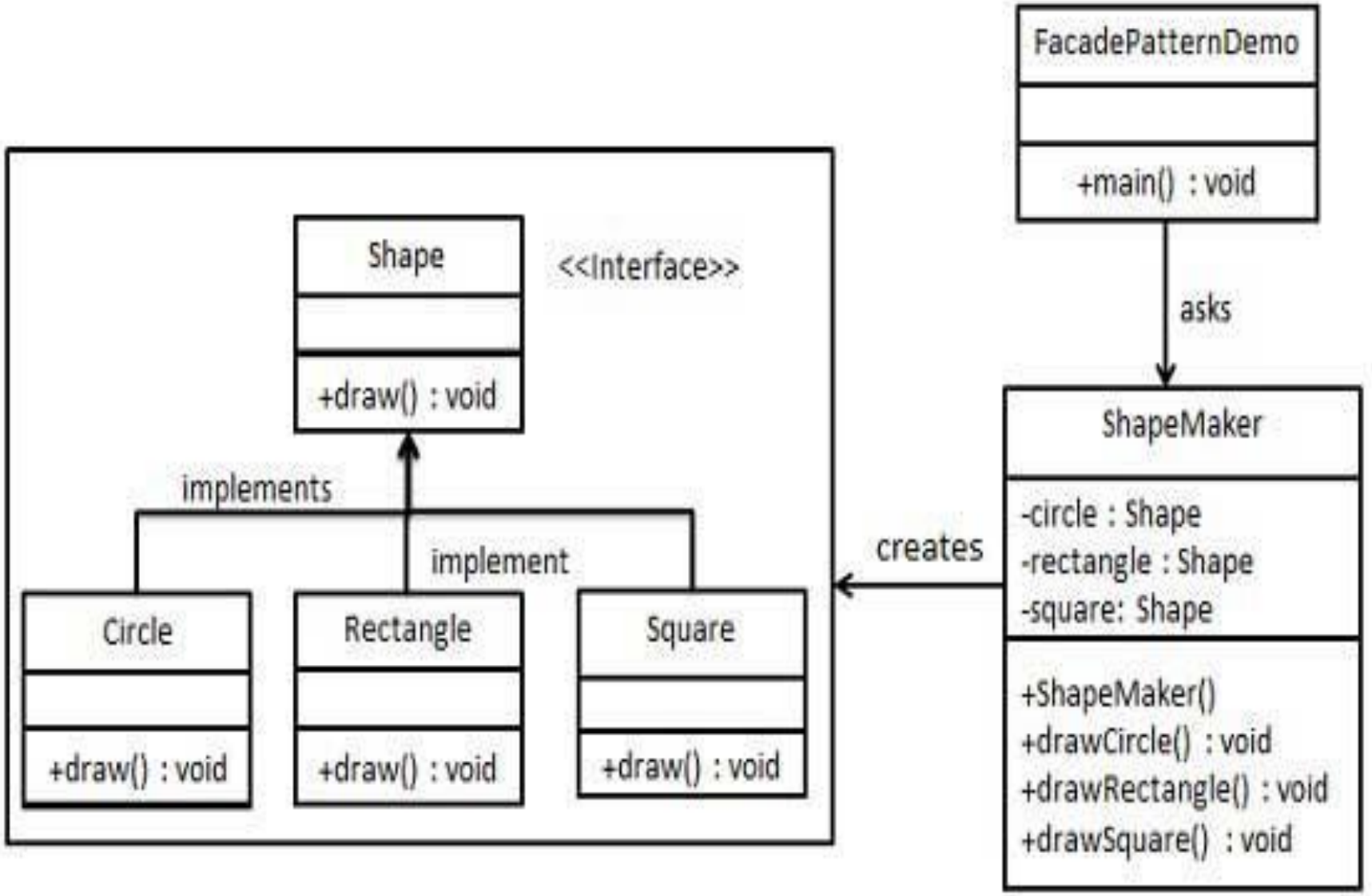
```
public class ShapeMaker {
private Shape circle;
private Shape rectangle;
private Shape square;

public ShapeMaker() {
circle = new Circle();
rectangle = new Rectangle();
square = new Square();
}

public void drawCircle(){
circle.draw();     }

public void drawRectangle(){
rectangle.draw();    }

public void drawSquare(){
square.draw();   }}
```

... Cont'd

Step 6:*Now, Creating a* ***client*** *that can draw the shapes from* ***shape*** *through* ***shapeMaker.***

```
public class Client {
public static void main(String[] args)
{
ShapeMaker shapeMaker = new ShapeMaker();
shapeMaker.drawCircle();
shapeMaker.drawRectangle();
shapeMaker.drawSquare();
}
}
```

… Cont'd


FacadePatternDemo
+main() : void
asks
ShapeMaker
-circle : Shape
-rectangle : Shape
-square: Shape
+ShapeMaker()
+drawCircle() : void
+drawRectangle() : void
+drawSquare() : void
creates
Shape
<<Interface>>
+draw() : void
implements
implement
Circle
+draw() : void
Rectangle
+draw() : void
Square
+draw() : void

Example2: Computer class acts as facade which encapsulates other complex classes

```
static class Program {
   static void Main() {
      Facade facade = new Facade();
      facade.PerformAction();
   }}
public class Façade {
   public void PerformAction()  {
      var c1a = new Class1A();
      var c1b = new Class1B();
      var c2a = new Class2A();
      var c2b = new Class2B();

      var result1a = c1a.Method1A();
      var result1b = c1b.Method1B(result1a);
      var result2a = c2a.Method2A(result1a);
      c2b.Method2B(result1b, result2a);
   }}
public class Class1A {
   public int Method1A()  {
      Console.WriteLine("Class1A.Method1A return value: 1");
      return 1;
   }}
```

```
public class Class1B {
   public int Method1B(int param)   {
      Console.WriteLine("Class1B.Method1B return value: {0}",param+1);
      return param+1; }
}

public class Class2A {
   public int Method2A(int param)
   {
      Console.WriteLine("Class2A.Method2A return value: {0}",param+2);
      return param+2;
   } }

public class Class2B {
   public void Method2B(int param1, int param2)
   {
      Console.WriteLine("Class2B.Method2B return value: {0}", param1+param2 );
   }
}
```

```
public class Computer {
   private readonly CPU _cpu;
   private readonly HardDrive _hardDrive;
   private readonly Memory _memory;

   private const long BootAddress = 1;
   private const long BootSector = 1;
   private const int SectorSize = 10;

   public Computer()   {
      _cpu = new CPU();
      _hardDrive = new HardDrive();
      _memory = new Memory();     }

   public void Start()
   {
      _cpu.Freeze();
      _memory.Load(BootAddress, _hardDrive.Read(BootSector, SectorSize));
      _cpu.Jump(BootAddress);
      _cpu.Execute();
   }}

public class CPU  {
   public void Freeze()  {
      Console.WriteLine("CPU is frozen");  }

   public void Jump(long position)  {
      Console.WriteLine("Jumping to position: {0}", position); }
```

```
public void Execute() {
      Console.WriteLine("Executing...");   }}

public class HardDrive {
      public byte[] Read(long lba, int size)   {
         var bytes = new byte[size];
         var random = new Random();
         random.NextBytes(bytes);
         return bytes;
      } }
public class Memory {
   public void Load(long position, byte[] data) {
      Console.WriteLine("Loading data: ");
      foreach (var b in data)  {
         Console.Write(b+ " ");
         Thread.Sleep(1000);   }

      Console.WriteLine("\nLoading compleded");   }}

class Program {
   static void Main() {
      var computer = new Computer();
      computer.Start();
   } }
```

Behavioral Design Patterns

- Behavioral design patterns help you define the communication between objects in your system and how the flow is controlled in a complex program.
- A behavioral pattern explains how objects interact.
- It describes how different objects and classes send messages to each other to make things happen and how the steps of a task are divided among different objects.
- Where Creational patterns mostly describe a moment of time (the instant of creation), and
- Structural patterns describe a more or less static structure, Behavioral patterns describe a process or a flow.

…Cont'd

- **The behavioral patterns are:**
  - Iterator
  - Chain of Responsibility
- **Advantages of Behavioral Design Patterns**
  - It reduces the complexity of communication between the objects.
  - It defines the best possible way to communicate between the objects.
  - It saves the number of resources.
  - By saving the number of resources, we can serve particular clients in a particular amount of time.

Use Case of Behavioral Design Pattern-

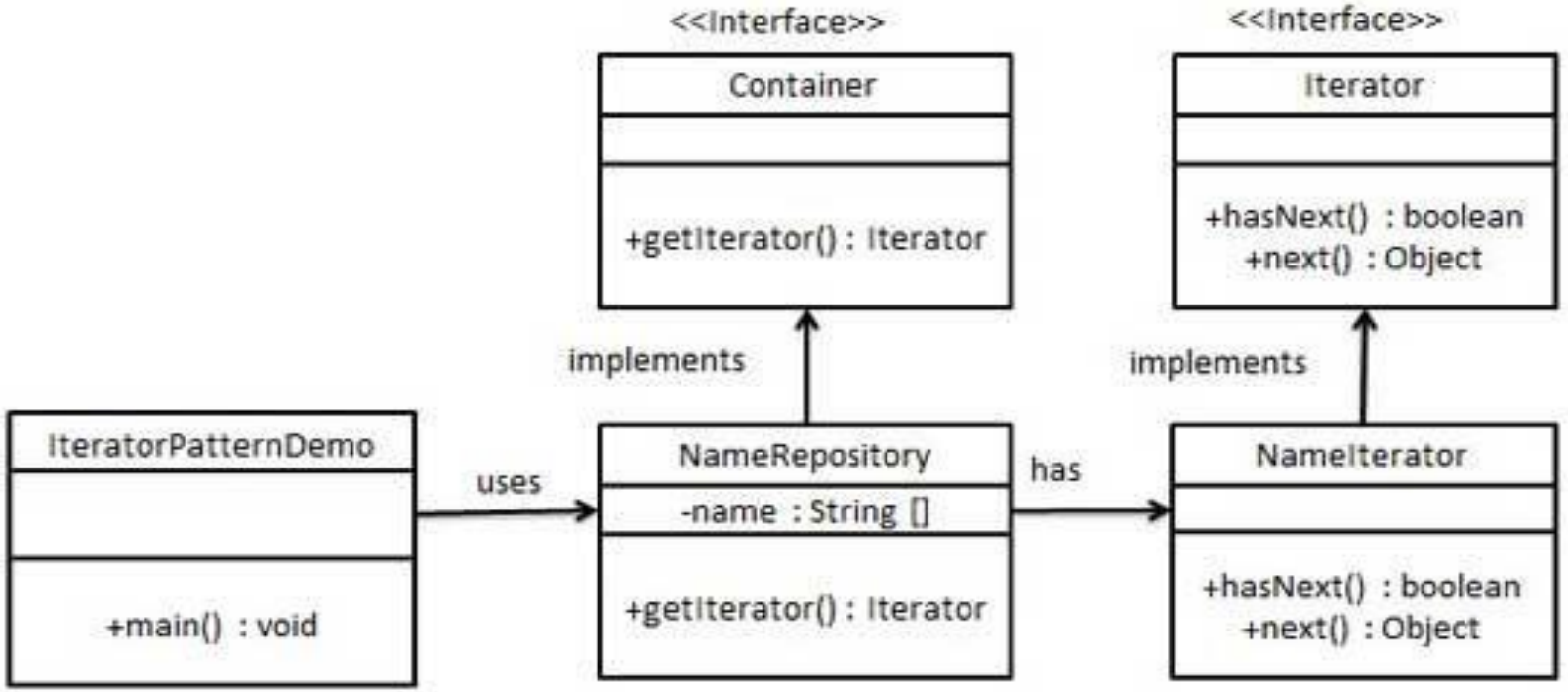
- The template pattern defines the skeleton of an algorithm in an operation deferring some steps to sub-classes.
- The template method lets subclasses redefine certain steps of an algorithm without changing the algorithm structure.
- For example, in your project, you want the behavior of the module to be able to extend, such that
  - we can make the module behave in new and different ways as the requirements of the application change, or to meet the needs of new applications.
  - However, no one is allowed to make source code changes to it,
  - i.e you can **add** but **can't modify** the structure in those scenarios a developer can approach template design pattern.

Iterator Pattern

- The key idea is to take the responsibility for access and traversal out of the list object and put it into an iterator object.
- Iterator Pattern is used **"to access the elements of** a collection **an aggregate object sequentially without exposing its underlying implementation".**

Usage:

- When you want to access a collection of objects without exposing its internal representation.
- When there are multiple traversals of objects need to be supported in the



… Cont'd

***Step 1**: Create a **Iterartor** interface*.

```
public interface Iterator {
public boolean hasNext();
public Object next();
}
```

***Step 2**: Create a* **interface** *interface.*

```
public interface Container {
public Iterator getIterator();
}
```

***Step 3:** Create a* **NameRepository** *class that will implement **Container** interface.*

```
public class NameRepository implements Container {
public String names[] = {"Robert" , "John" ,"Julie" , "Lora"};

public Iterator getIterator() {
return new NameIterator();
}
```

… Cont’d

```
private class NameIterator implements Iterator {
int index;

public boolean hasNext() {
if(index < names.length){
return true;
}
return false;
}

public Object next() {
if(this.hasNext()){
return names[index++];
}
return null;
}
} }
```

Step 4: *Create a* ***IteratorPatternDemo*** *class.*

File: IteratorPatternDemo.java

```
public class IteratorPatternDemo {
   public static void main(String[] args) {
      CollectionofNames cmpnyRepository = new CollectionofNames();


      for(Iterator iter = cmpnyRepository.getIterator(); iter.hasNext();){

         String name = (String)iter.next();
         System.out.println("Name : " + name);
       }
   }
}
```

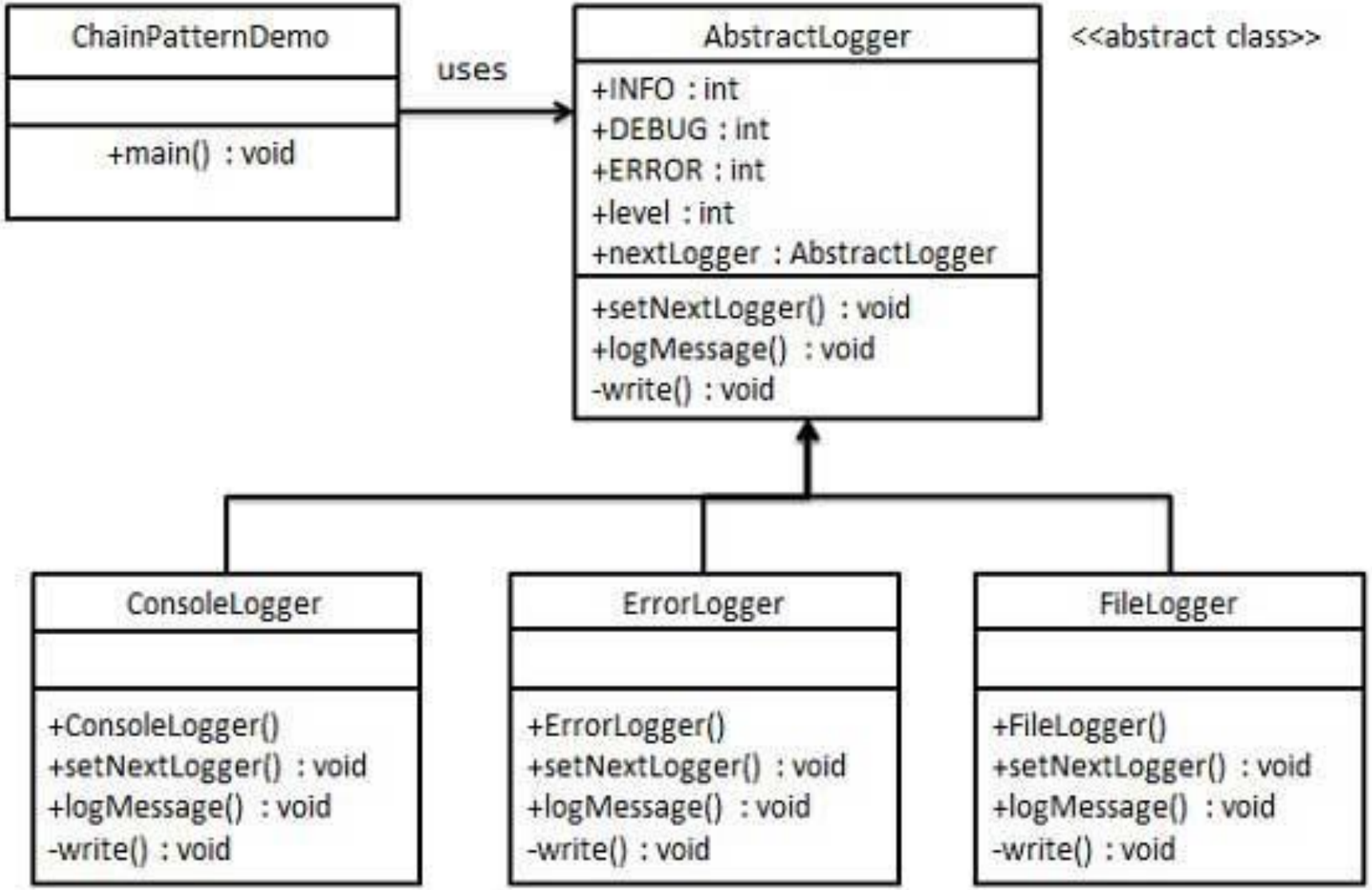
Chain of Responsibility

- Chain of responsibility pattern **creates a chain of receiver objects for a request.**
- In this pattern, normally each receiver contains reference to another receiver.
- **If one object cannot handle the request then it passes the same to the next receiver and so on.**
- Chain of Responsibility allows a number of classes to attempt to handle a request, independently of any other object along the chain.
- In this pattern, the object which sends a command doesn't know which object will process the command.
- The chain of responsibility pattern is a design pattern that defines a linked list of handlers, each of which is able to process requests.
  - When a request is submitted to the chain, it is passed to the first handler in the list that is able to process it.

… Cont’d

- For example, an ATM uses the Chain of Responsibility design pattern in money giving process.
- Advantage of Chain of Responsibility Pattern
  - It reduces the coupling.
  - It adds flexibility while assigning the responsibilities to objects.
  - It allows a set of classes to act as one; events produced in one class can be sent to other handler classes with the help of composition.
- Usage of Chain of Responsibility Pattern:
  - When more than one object can handle a request and the handler is unknown.
  - When the group of objects that can handle the request must be specified in dynamic way.

Cont'd…


ChainPatternDemo
+main() : void
uses
AbstractLogger
<<abstract class>>
+INFO : int
+DEBUG : int
+ERROR : int
+level : int
+nextLogger : AbstractLogger
+setNextLogger() : void
+logMessage() : void
-write() : void
ConsoleLogger
+ConsoleLogger()
+setNextLogger() : void
+logMessage() : void
-write() : void
ErrorLogger
+ErrorLogger()
+setNextLogger() : void
+logMessage() : void
-write() : void
FileLogger
+FileLogger()
+setNextLogger() : void
+logMessage() : void
-write() : void

Chain of Responsibility

Step 1: *Create a* ***Logger*** *abstract class.*

```
public abstract class AbstractLogger {
public static int INFO = 1;    public static int DEBUG = 2;
public static int ERROR = 3;     protected int level;
//next element in chain or responsibility
protected AbstractLogger nextLogger;
public void setNextLogger(AbstractLogger nextLogger){
this.nextLogger = nextLogger;
}
public void logMessage(int level, String message){
if(this.level <= level){
write(message);
}
if(nextLogger !=null){
nextLogger.logMessage(level, message);
} }
abstract protected void write(String message);
}
```

……Chain of Responsibility

***Step 2**: Create a **ConsoleBasedLogger** class.*

```
public class ConsoleLogger extends AbstractLogger {
public ConsoleLogger(int level){
this.level = level;
}

protected void write(String message) {
System.out.println("Standard Console::Logger: " + message);
}}
```

***Step 3**:* Create a **ErrorLogger** class.

```
public class ErrorLogger extends AbstractLogger {
public ErrorLogger(int level){
this.level = level;
}

protected void write(String message) {
System.out.println("Error Console::Logger: " + message);
}
}
```

….Chain of Responsibility

***Step 3**: Create a* **FileLogge**r *class.*

```
public class FileLogger extends AbstractLogger {
public FileLogger(int level){
this.level = level;
}
protected void write(String message) {
System.out.println("File::Logger: " + message);
}
}
```

.....Chain of Responsibility

```
public class ChainPatternDemo {
private static AbstractLogger getChainOfLoggers(){
AbstractLogger errorLogger = new ErrorLogger(AbstractLogger.ERROR);
AbstractLogger fileLogger = new FileLogger(AbstractLogger.DEBUG);
AbstractLogger consoleLogger = new ConsoleLogger(AbstractLogger.INFO);
errorLogger.setNextLogger(fileLogger);
fileLogger.setNextLogger(consoleLogger);
return errorLogger;
}

public static void main(String[] args) {
AbstractLogger loggerChain = getChainOfLoggers();
loggerChain.logMessage(AbstractLogger.INFO, "This is an information.");
loggerChain.logMessage(AbstractLogger.DEBUG,
"This is a debug level information.");
loggerChain.logMessage(AbstractLogger.ERROR, "This is an error information.");
} }
```

Reading Assignment

- **Creational Patterns**
  - Builder
  - Prototype
- **Structural Patterns**
  - Bridge
  - Composite
  - Decorator
  - Flyweight
  - Proxy
- **Behavioral Patterns**
  - Interpreter
  - Template
  - Command
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Visitor

Thank You !!!
Questions?