# Design Patterns

BY:Yibeltal A.

2012 E.C

# Introduction

- **Design patterns capture the *best practices of experienced object-oriented software developers*.**
- **Design patterns are solutions to general software development problems.**
- 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.

# Pattern's Elements

- In general, a pattern has four essential elements.
  - The **pattern name**
  - The **problem**
  - The **solution**
- The **pattern name** is a handle we can use to describe a design problem, its solutions, and consequences in a word or two.
- Naming a pattern immediately increases the design vocabulary.
- It lets us design at a higher level of abstraction.
- It makes it easier to think about designs and to communicate them and their tradeoffs to others.

# Cont..

- The **problem** describes when to apply the pattern.
  - It explains the problem and its context.
  - It might describe specific design problems such as how to represent algorithms as objects.
  - It might describe class or object structures that are symptomatic of an inflexible design.
  - Sometimes the problem will include a list of conditions that must be met before it makes sense to apply the pattern.

# Cont…

- The **solution** describes the elements that make up the design, their relationships, responsibilities and collaborations.
- The solution doesn't describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many different situations.
- Instead, the pattern provides an abstract description of a design problem and how a general arrangement of elements (classes and objects in our case) solves it.

# Types of Design Patterns

- Design patterns are divided into **three** types:
  - ***Creational patterns*** are ones that create objects for you, rather than having you instantiate objects directly. This gives your program more flexibility in deciding which objects need to be created for a given case.
  - ***Structural patterns*** help you compose groups of objects into larger structures, such as complex user interfaces or accounting data.
  - ***Behavioral patterns*** help you define the communication between objects in your system and how the flow is controlled in a complex program.

# Creational patterns

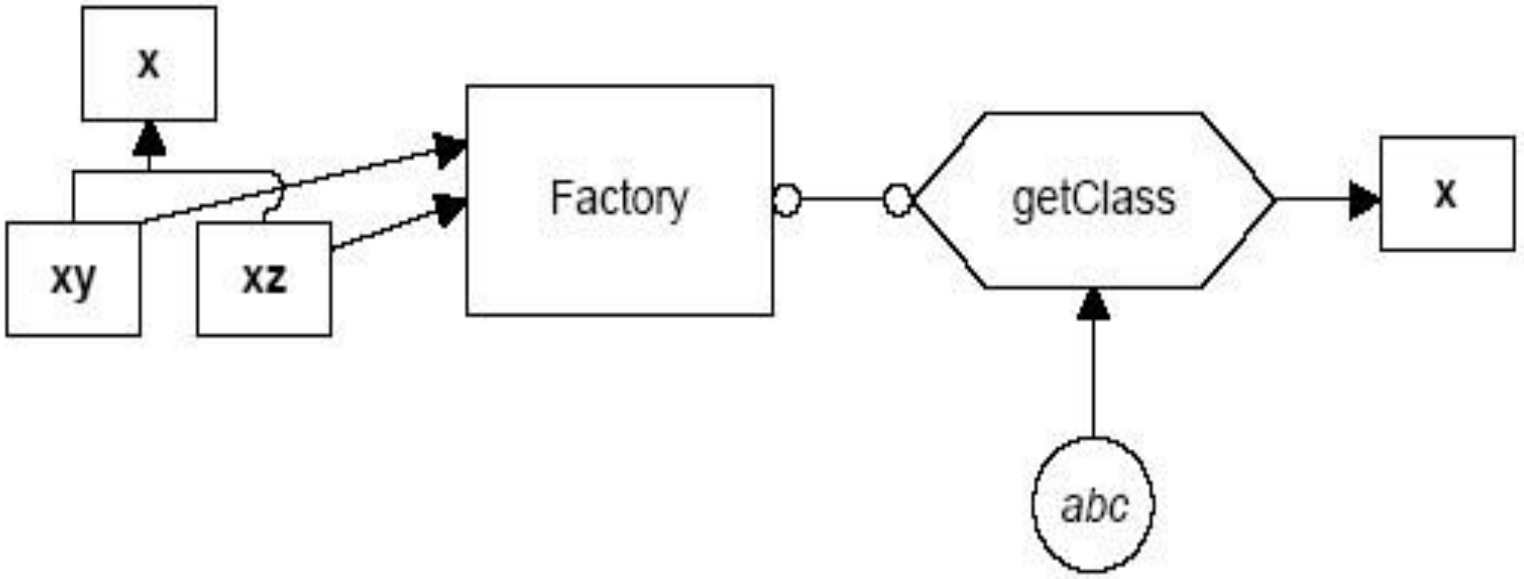
- The creational patterns deal with the best way to **create instances of objects**.
- In Java, the simplest way to create an instance of an object is by using the **new** operator.
- ***Student = new Student();*** //instance of Student class
- This amounts to hard coding, depending on how you create the object within your program.
- 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.

# Cont…

- **The Factory Pattern** provides a simple decision making class that returns one of several possible subclasses of an abstract base class depending on the data that are provided.
- **The Abstract Factory Pattern** provides an interface to create and return one of several families of related objects.
- **The Singleton Pattern** is a class of which there can be no more than one instance.
- It provides a single global point of access to that instance.

# The Factory Pattern

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



# Cont…

- 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…

```
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.");
}
}
```

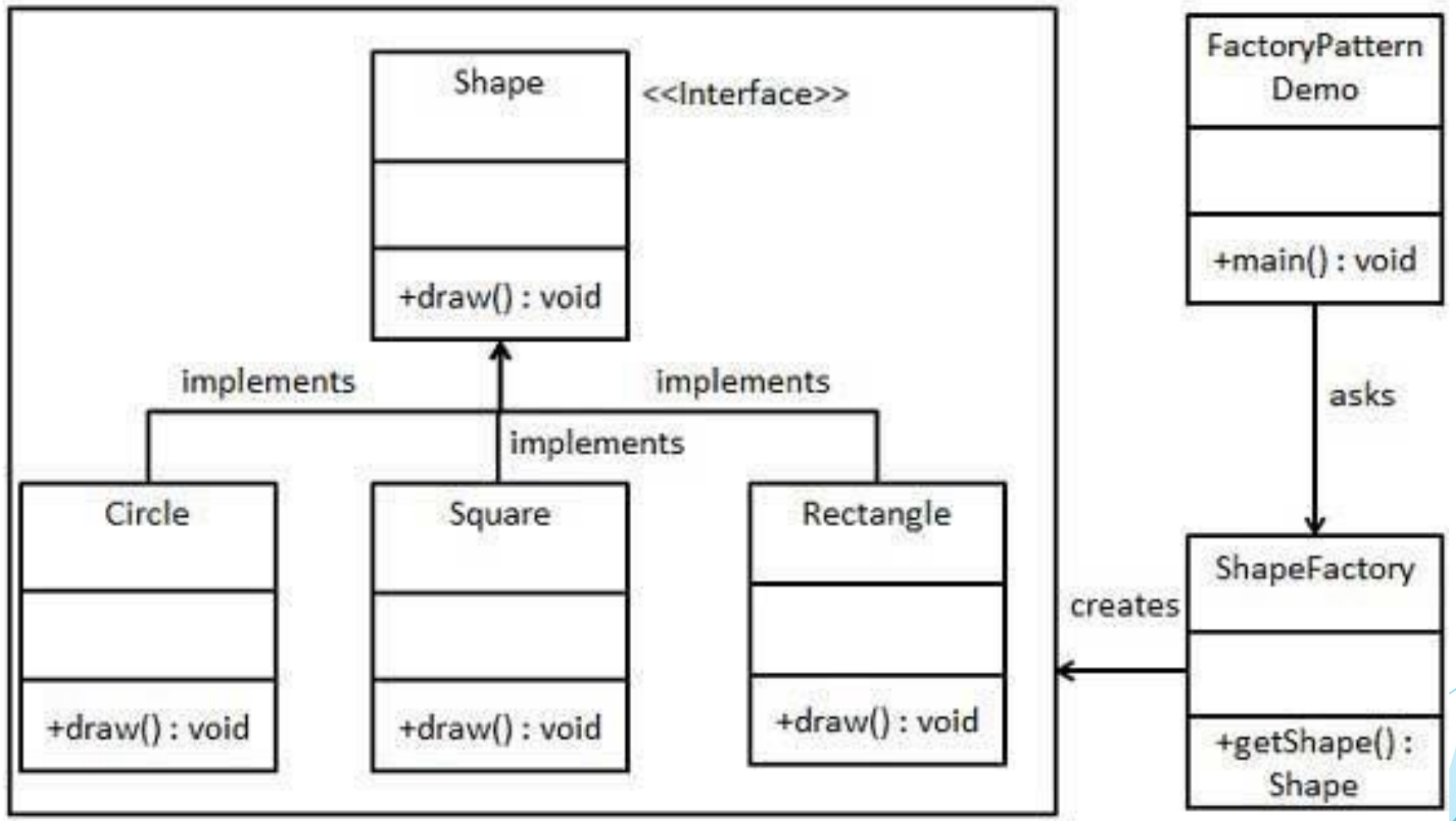
# Cont…

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

# Cont…

```
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…


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

# Cont…

```
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();

} }
```

# The Abstract Factory Pattern

- The Abstract Factory pattern is one level of abstraction higher than the factory pattern.
- This pattern returns one of several related classes, each of which can return several different objects on request.
- In other words, **the Abstract Factory is a factory object that returns one of several factories.**
- One classic application of the abstract factory is the case where your system needs to support multiple “look-and-feel” user interfaces, such as Windows, Motif or Macintosh:

# Cont…

- You tell the factory that you want your program to look like Windows and it returns a GUI factory which returns Windows-like objects.
- When you request specific objects such as buttons, check boxes and windows, the GUI factory returns Windows instances of these visual interface components.

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

# Cont…

```
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.");
} }
```

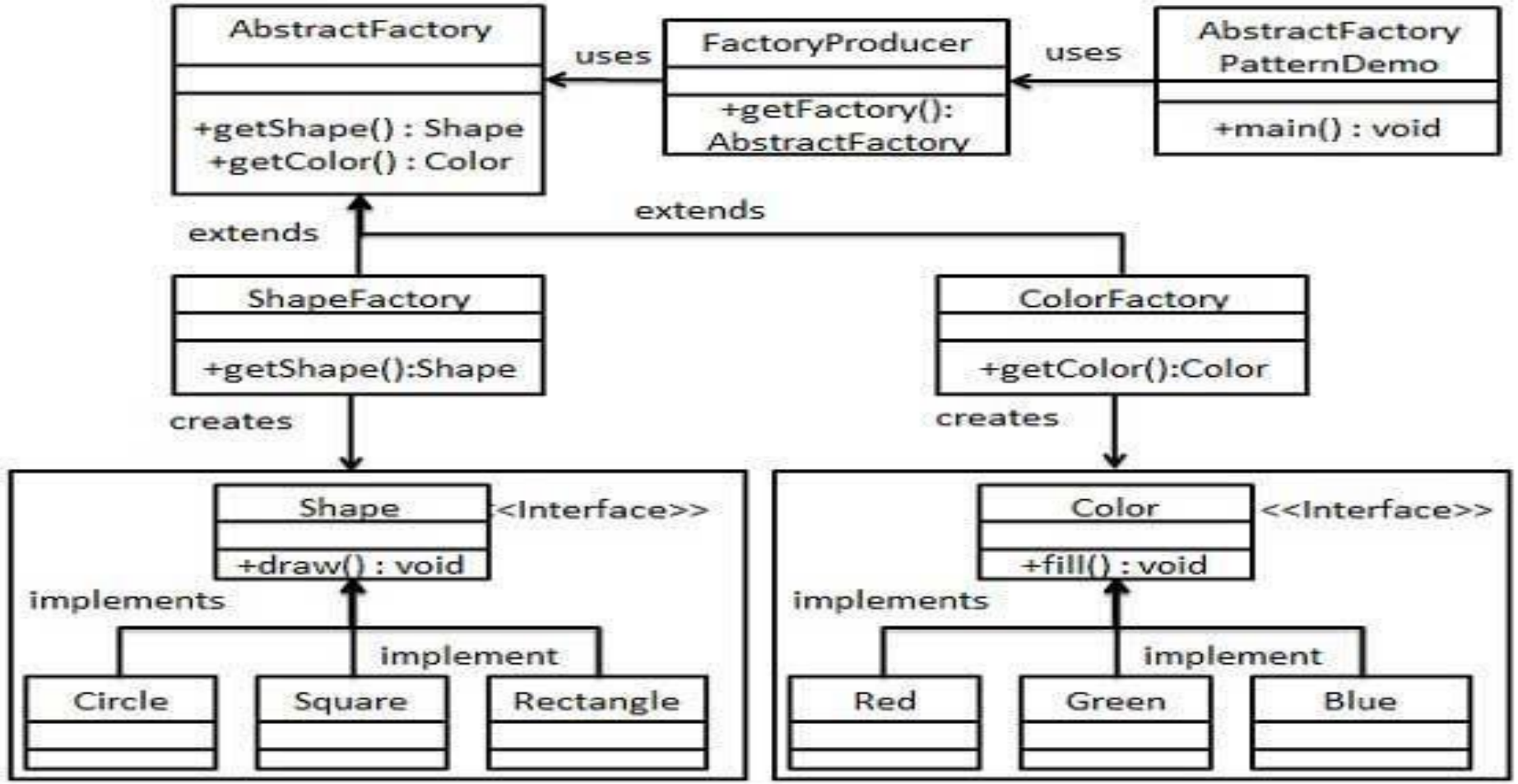
# Cont…

```
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…

```
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…


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…

```
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….

- 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.

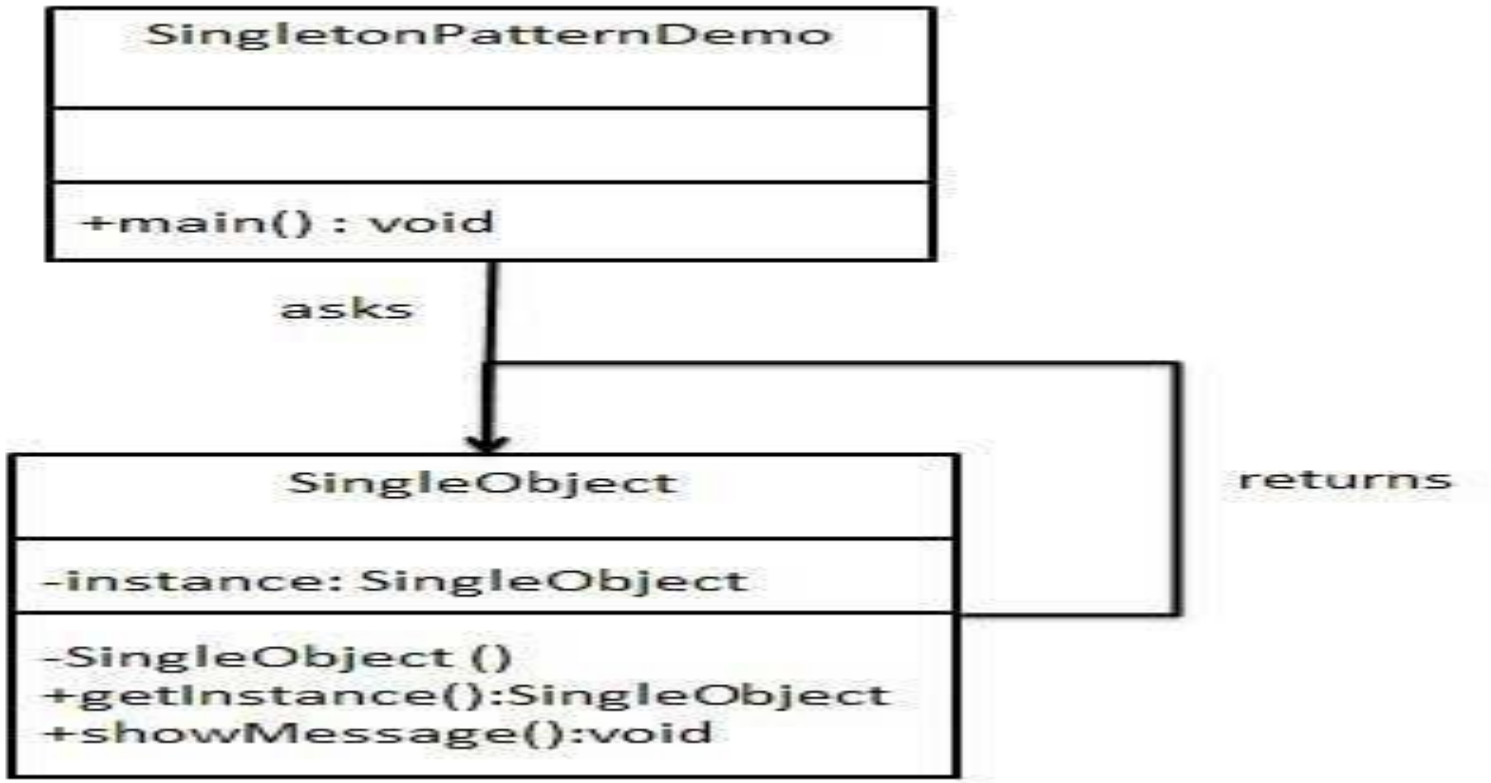
# Singleton Pattern

- Sometimes it is appropriate to have exactly one instance of a class: print spoolers, file systems.
- Typically, those types of objects known as singletons, are accessed by disparate objects throughout a software system, and therefore require a global point of access.
- The Singleton pattern addresses all the concerns above.
- With the Singleton design pattern you can:
- Ensure that only one instance of a class is created.
- Provide a global point of access to the object.

# Cont…

- The Singleton pattern ensures a class has only one instance, and provides a global point of access to it.
- The class itself is responsible for keeping track of its sole instance.
- 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…


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

# Cont..

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

# Cont…

- The *ClassicSingleton* class maintains a static reference to the alone singleton instance and returns that reference from the static getInstance() method.
- The ClassicSingleton class employs a technique known as lazy instantiation to create the singleton; 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.
- **The ClassicSingleton class is not thread-safe.** Solution: **Synchronization**

# Cont…..

```
public class ClassicSingleton { private static ClassicSingleton
instance = null; private static Object syncObject; // to
synchronize a block private ClassicSingleton() {

/*exists only to defeat instantiation*/ }; public static
ClassicSingleton getInstance() { synchronized(syncObject) {

if (instance == null) instance = new

ClassicSingleton();} return instance;}

}
```

# Cont…

- 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.
- The Structural patterns are:
  - Adapter
  - Façade

# Adapter Pattern

- Adapter pattern works as a connector between two incompatible interfaces.
- This type of design pattern comes under structural pattern as this pattern combines the capability of two independent interfaces.
- This pattern involves a single class which is responsible to join functionalities of independent 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.

# Cont…

- We have a MediaPlayer interface and a concrete class AudioPlayer implementing theMediaPlayer interface.
- AudioPlayer can play mp3 format audio files by default.
- We are having another interface AdvancedMediaPlayer and concrete classes implementing the AdvancedMediaPlayer interface. § These classes can play vlc and mp4 format files.
- We want to make AudioPlayer to play other formats as well.

# Cont…

- To attain this, we have created an adapter class MediaAdapter which implements the MediaPlayer interface and uses AdvancedMediaPlayer objects to play the required format.
- AudioPlayer uses the adapter class MediaAdapter passing it the desired audio type without knowing the actual class which can play the desired format.
- AdapterPatternDemo, our demo class, will use AudioPlayer class to play various formats.

# Cont…

public interface MediaPlayer { public void play(String audioType, String fileName);

}

public interface AdvancedMediaPlayer { public void playVlc(String fileName);

public void playMp4(String fileName);

}

# Cont…

```
public class VlcPlayer implements AdvancedMediaPlayer{
public void playVlc(String fileName) {
System.out.println("Playing vlc file. Name: "+ fileName);
}
public void playMp4(String fileName) {
//do nothing
} }
```

# Cont…

```
public class Mp4Player implements AdvancedMediaPlayer{
public void playVlc(String fileName) {
//do nothing
}
public void playMp4(String fileName) {
System.out.println("Playing mp4 file. Name: "+ fileName);
}
}
```

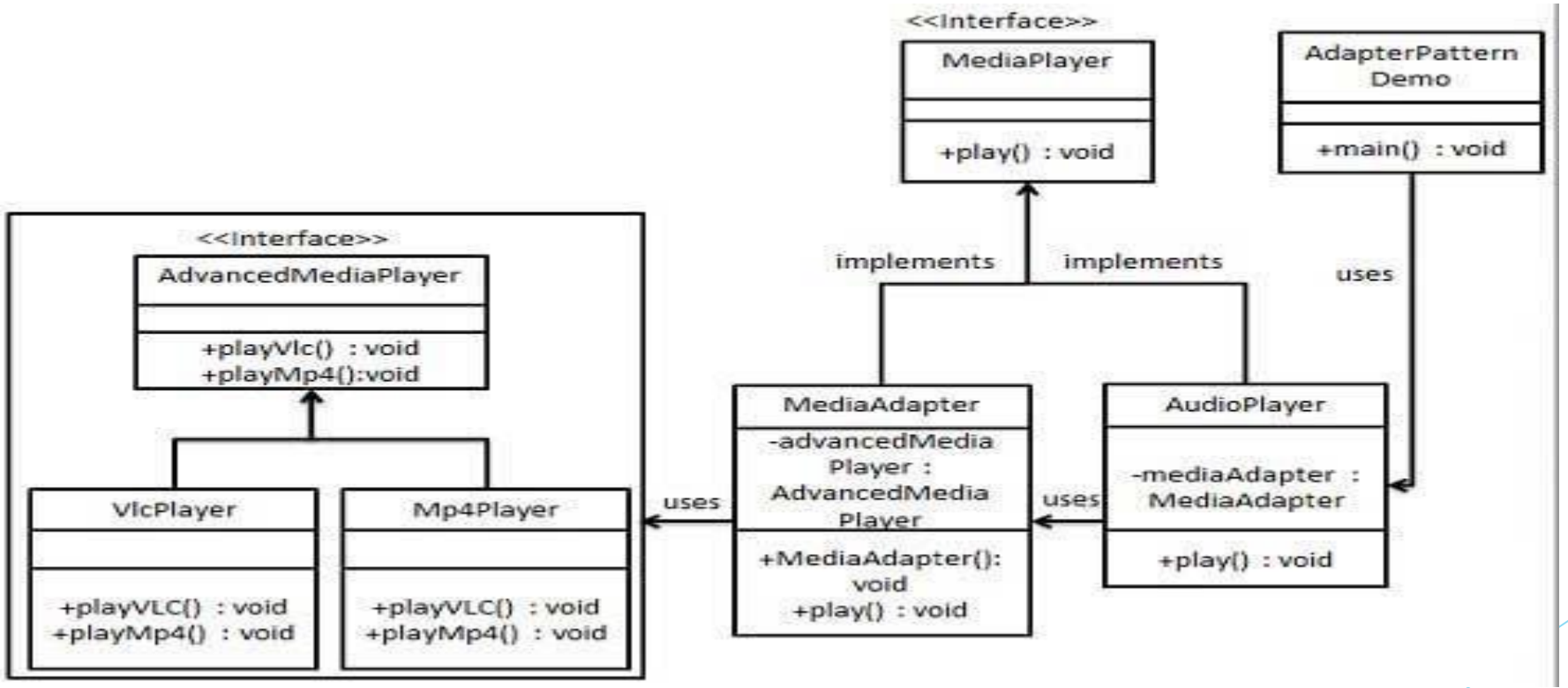
# Cont..

```
public class MediaAdapter implements MediaPlayer { AdvancedMediaPlayer
advancedMusicPlayer;

public MediaAdapter(String audioType){ if(audioType.equalsIgnoreCase("vlc") ){
advancedMusicPlayer = new VlcPlayer(); } else if (audioType.equalsIgnoreCase("mp4")){
advancedMusicPlayer = new Mp4Player();

}

}

public void play(String audioType, String fileName) {
if(audioType.equalsIgnoreCase("vlc")){ advancedMusicPlayer.playVlc(fileName); }else
if(audioType.equalsIgnoreCase("mp4")){ advancedMusicPlayer.playMp4(fileName);

} } }
```

# Cont..

```
public class AudioPlayer implements MediaPlayer { MediaAdapter mediaAdapter;
public void play(String audioType, String fileName) {
if(audioType.equalsIgnoreCase("mp3")){
System.out.println("Playing mp3 file. Name: "+ fileName);
}
//mediaAdapter is providing support to play other file formats else
if(audioType.equalsIgnoreCase("vlc") || audioType.equalsIgnoreCase("mp4")){ mediaAdapter =
new MediaAdapter(audioType); mediaAdapter.play(audioType, fileName);
}
else{
System.out.println("Invalid media. "+
audioType + " format not supported");
} }
}
```

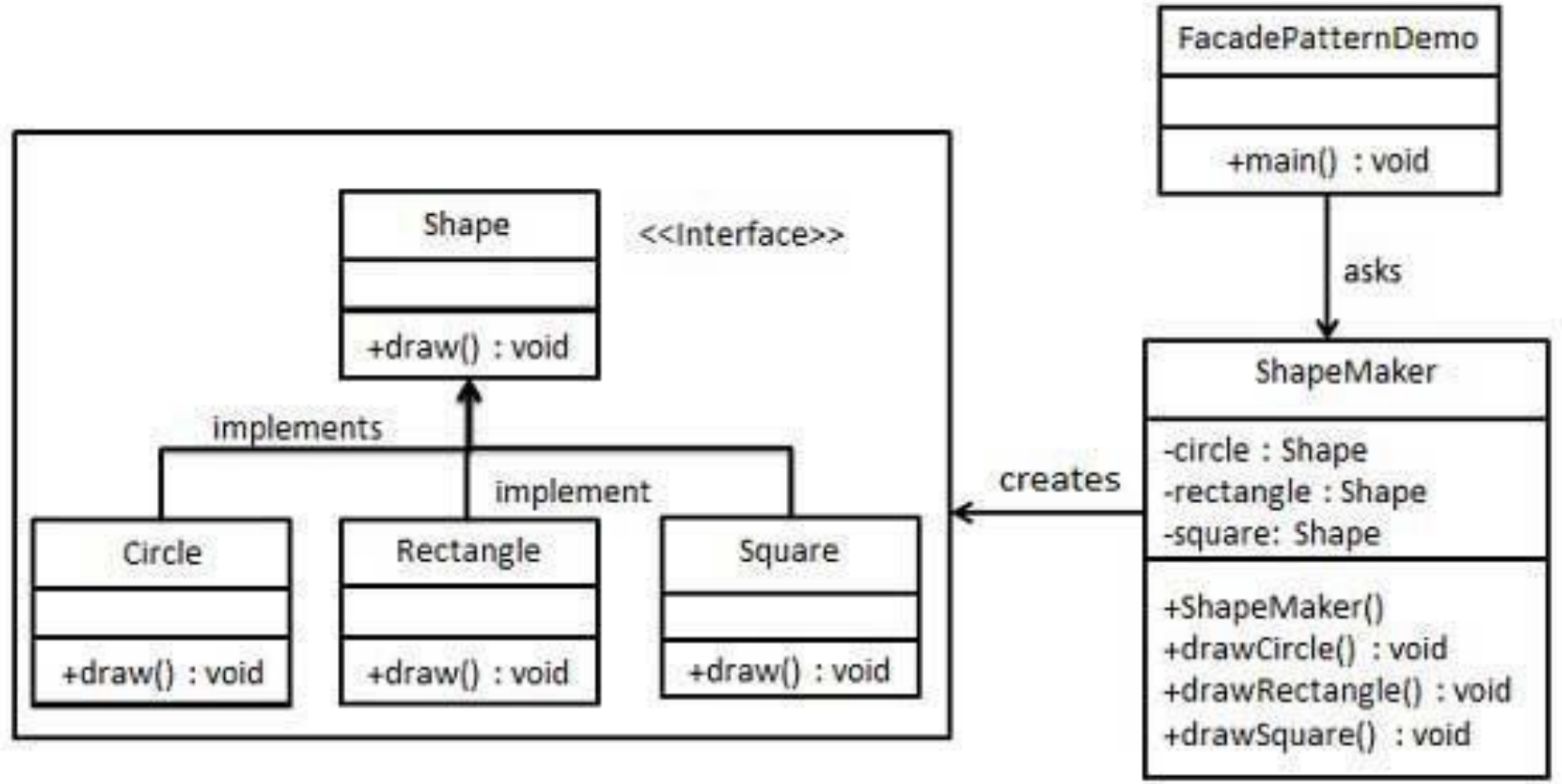
# Cont…


<<Interface>>
MediaPlayer
+play() : void
AdapterPattern Demo
+main() : void
implements
implements
uses
<<Interface>>
AdvancedMediaPlayer
+playVlc() : void
+playMp4():void
MediaAdapter
-advancedMedia Player : AdvancedMedia Player
+MediaAdapter(): void
+play() : void
AudioPlayer
-mediaAdapter : MediaAdapter
+play() : void
uses
uses
VlcPlayer
+playVLC() : void
+playMp4() : void
Mp4Player
+playVLC() : void
+playMp4() : void

# Facade Pattern

- Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system.
- This type of design pattern comes under structural pattern as this pattern adds an interface to existing system to hide its complexities.
- This pattern involves a single class which provides simplified methods required by client and delegates calls to methods of existing system classes.

# Cont….


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

# Cont…

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

# Cont…

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

# Cont…

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

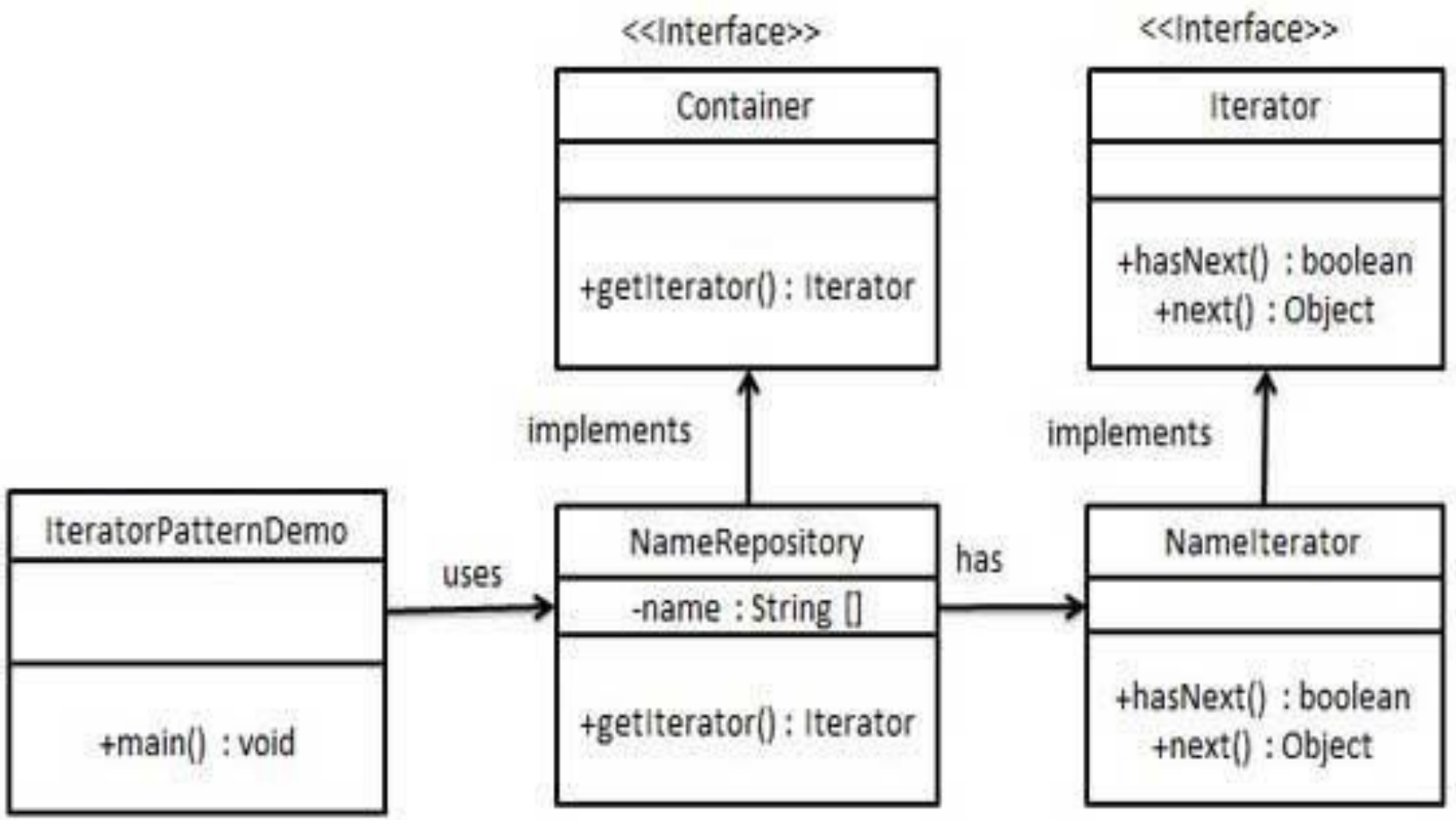
# Behavioral Design Patterns

- Behavioral design patterns help you define the communication between objects in your system and how the flow is controlled in a complex program.
  - The behavioral patterns are:
  - Iterator
  - Chain of Responsibility

# Iterator Pattern

- An aggregate object such as a list should give you a way to access its elements without exposing its internal structure.
- The key idea is to take the responsibility for access and traversal out of the list object and put it into an iterator object.
- This pattern is used to get a way to access the elements of a collection object in sequential manner without any need to know its underlying representation.

# Cont…


<<Interface>>
Container
+getIterator() : Iterator
<<Interface>>
Iterator
+hasNext() : boolean
+next() : Object
implements
implements
IteratorPatternDemo
+main() : void
uses
NameRepository
-name : String []
+getIterator() : Iterator
has
NameIterator
+hasNext() : boolean
+next() : Object

# Cont…

```
public interface Iterator { public boolean hasNext();
public Object next();
}
public interface Container {
public Iterator getIterator();
}
public class NameRepository implements Container { public String names[] = {"Robert" ,
"John" ,"Julie" , "Lora"};
public Iterator getIterator() {
return new NameIterator();
}
```

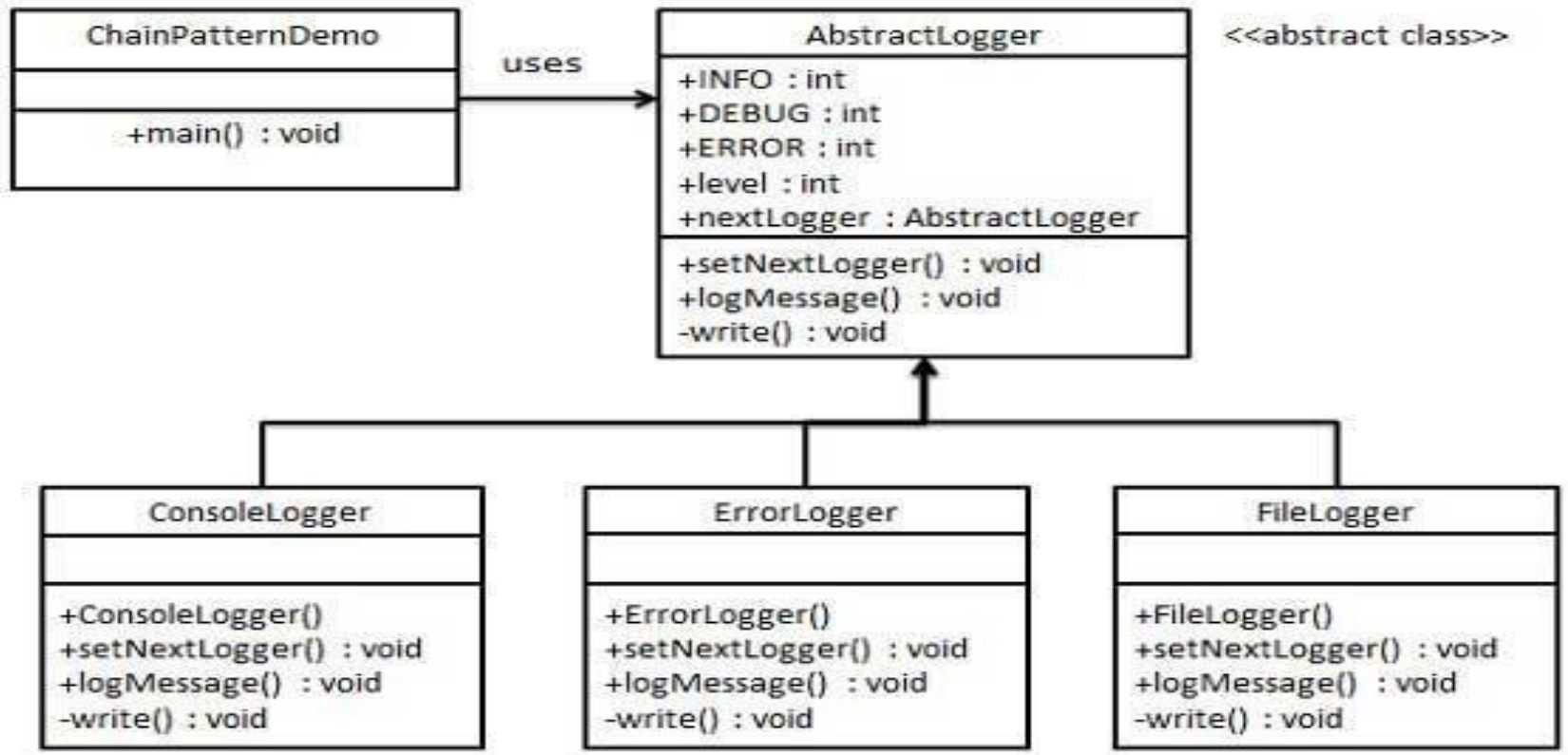
# Cont…

```
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;
}
} }
```

# Cont…

- Chain of responsibility pattern creates a chain of receiver objects for a request.
- This pattern decouples sender and receiver of a request based on type of 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.
- Once the request is handled, it completes it's journey through the chain.

# Cont…


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

# Cont…

```
public class  ConsoleLogger extends AbstractLogger {
public ConsoleLogger(int level){
this.level = level;
}
protected void write(String message) {
System.out.println("Standard Console::Logger: " + message);
}}
public class ErrorLogger extends AbstractLogger {
public ErrorLogger(int level){
this.level = level;
}
protected void write(String message) {
System.out.println("Error Console::Logger: " + message);
}
}
```

# Cont….

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

# Cont…

```
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.");

} }
```

Thanks