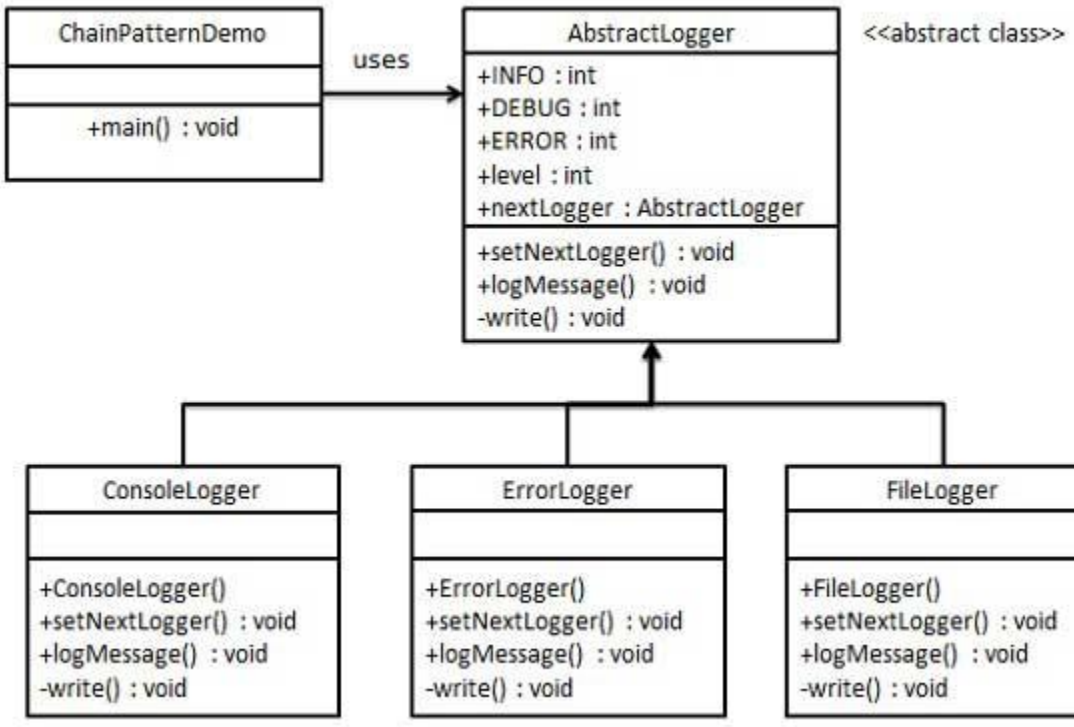
Chain of responsibility pattern:

As the name suggests, the 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. This pattern comes under behavioral patterns.

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.

Implementation

We have created an abstract class *AbstractLogger* with a level of logging. Then we have created three types of loggers extending the *AbstractLogger*. Each logger checks the level of message to its level and print accordingly otherwise does not print and pass the message to its next logger.



Step 1

Create an abstract logger class.

AbstractLogger.java

```
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);

}
```

Step 2

Create concrete classes extending the logger.

ConsoleLogger.java

```
public class ConsoleLogger extends AbstractLogger {

   public ConsoleLogger(int level){
      this.level = level;
   }

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

ErrorLogger.java

```
public class ErrorLogger extends AbstractLogger {

   public ErrorLogger(int level){
      this.level = level;
   }

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

FileLogger.java

```
public class FileLogger extends AbstractLogger {

   public FileLogger(int level){
      this.level = level;
   }

   @Override
   protected void write(String message) {
      System.out.println("File::Logger: " + message);
   }
}
```

Step 3

Create different types of loggers. Assign them error levels and set next logger in each logger. Next logger in each logger represents the part of the chain.

ChainPatternDemo.java

```
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 an debug level information.");

    loggerChain.logMessage(AbstractLogger.ERROR,
      "This is an error information.");
  }
}
```

Step 4

Verify the output.

```
Standard Console::Logger: This is an information.
File::Logger: This is an debug level information.
Standard Console::Logger: This is an debug level information.
Error Console::Logger: This is an error information.
File::Logger: This is an error information.
Standard Console::Logger: This is an error information.
```

State pattern:

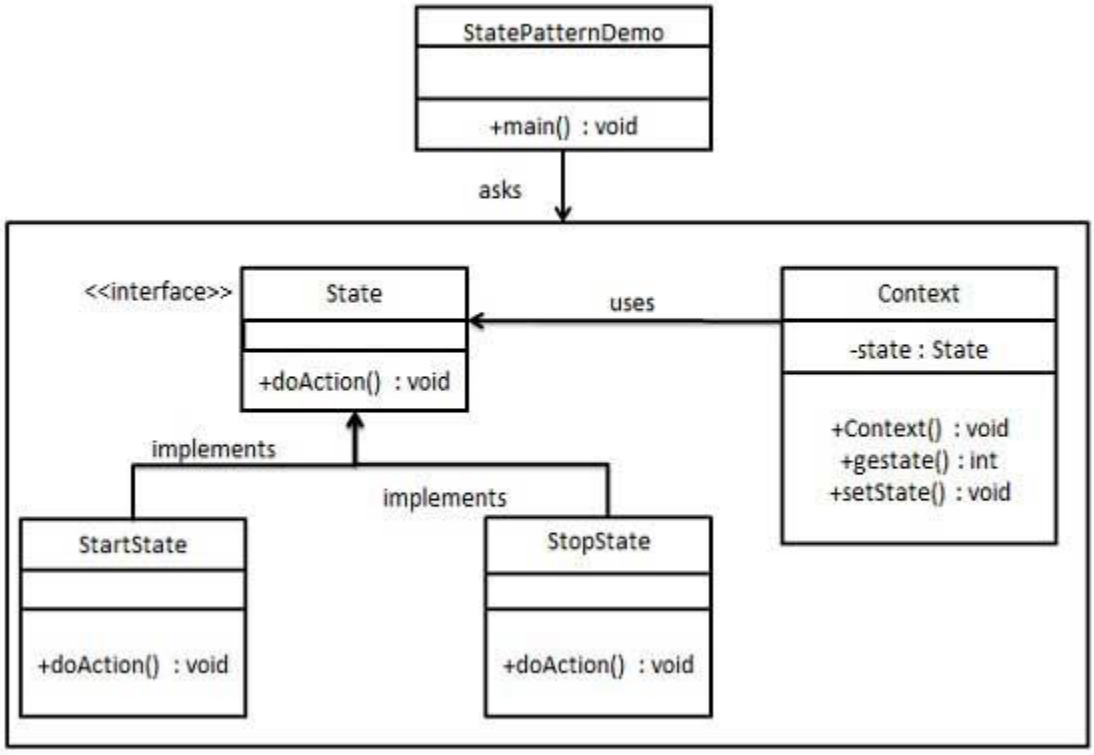
In State pattern a class behaviour changes based on its state. This type of design pattern comes under behaviour pattern.

In State pattern, we create objects which represent various states and a context object whose behaviour varies as its state object changes.

Implementation

We are going to create a *State* interface defining an action and concrete state classes implementing the *State* interface. *Context* is a class which carries a State.

StatePatternDemo, our demo class, will use *Context* and state objects to demonstrate change in Context behavior based on type of state it is in.



Step 1

Create an interface.

State.java

```
public interface State {
   public void doAction(Context context);
}
```

Step 2

Create concrete classes implementing the same interface.

StartState.java

```
public class StartState implements State {

   public void doAction(Context context) {
      System.out.println("Player is in start state");
      context.setState(this);
   }

   public String toString(){
      return "Start State";
   }
}
```

StopState.java

```
public class StopState implements State {

   public void doAction(Context context) {
      System.out.println("Player is in stop state");
      context.setState(this);
   }

   public String toString(){
      return "Stop State";
   }
}
```

Step 3

Create *Context* Class.

Context.java

```
public class Context {
  private State state;

  public Context(){
    state = null;
  }

  public void setState(State state){
    this.state = state;
  }

  public State getState(){
    return state;
  }
}
```

Step 4

Use the *Context* to see change in behaviour when *State* changes.

StatePatternDemo.java

```
public class StatePatternDemo {
  public static void main(String[] args) {
    Context context = new Context();

    StartState startState = new StartState();
    startState.doAction(context);

    System.out.println(context.getState().toString());

    StopState stopState = new StopState();
    stopState.doAction(context);

    System.out.println(context.getState().toString());
  }
}
```

Step 5

Verify the output.

```
Player is in start state
Start State
Player is in stop state
Stop State
```

Iterator pattern:

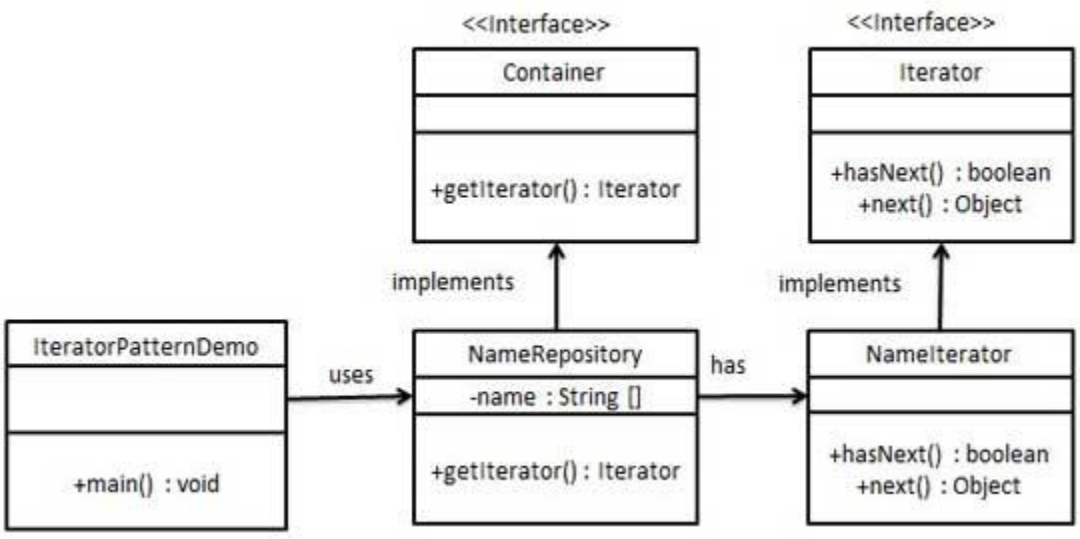
Iterator pattern is very commonly used design pattern in Java and .Net programming environment. This pattern is used to get a way to access the elements of a collection object in sequential manner without any need to know its underlying representation.

Iterator pattern falls under behavioural pattern category.

Implementation

We're going to create a *Iterator* interface which narrates navigation method and a *Container* interface which returns the iterator. Concrete classes implementing the *Container* interface will be responsible to implement *Iterator* interface and use it.

IteratorPatternDemo, our demo class will use *NamesRepository*, a concrete class implementation to print a *Names* stored as a collection in *NamesRepository*.



Step 1

Create interfaces.

Iterator.java

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

Container.java

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

Step 2

Create concrete class implementing the *Container* interface. This class has inner class *NameIterator* implementing the *Iterator* interface.

NameRepository.java

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

   @Override
   public Iterator getIterator() {
      return new NameIterator();
   }

   private class NameIterator implements Iterator {

      int index;

      @Override
      public boolean hasNext() {

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

      @Override
```

```
    public Object next() {

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

Step 3

Use the *NameRepository* to get iterator and print names.

IteratorPatternDemo.java

```
public class IteratorPatternDemo {

  public static void main(String[] args) {
    NameRepository namesRepository = new NameRepository();

    for(Iterator iter = namesRepository.getIterator(); iter.hasNext();){
      String name = (String)iter.next();
      System.out.println("Name : " + name);
    }
  }
}
```

Step 4

Verify the output.

```
Name : Robert
Name : John
Name : Julie
Name : Lora
```