If we have to change the behavior of an object based on its state, we can have a state variable in the Object. Then use **if-else** condition block to perform different actions based on the state. State design pattern is used to provide a systematic and loosely coupled way to achieve this through Context and State implementations.

State Pattern **Context** is the class that has a State reference to one of the concrete implementations of the State. Context forwards the request to the state object for processing. Let’s understand this with a simple example.

Suppose we want to implement a TV Remote with a simple button to perform action. If the State is ON, it will turn on the TV and if state is OFF, it will turn off the TV.

We can implement it using if-else condition like below;

class TVRemoteBasic{

private String state = "";

public void setState(String state) {

this.state = state;

}

public void doAction() {

if(state.equalsIgnoreCase("ON")) {

System.out.println("TV turned ON");

}else {

System.out.println("TV turen OFF");

}

}

}

public class TestStatePattern {

public static void main(String[] args) {

TVRemoteBasic remote = new TVRemoteBasic();

remote.setState("ON");

remote.doAction();

remote.setState("OFF");

remote.doAction();

}

}

Notice that client code should know the specific values to use for setting the state of remote. Further more if number of states increase then the tight coupling between implementation and the client code will be very hard to maintain and extend.

Now we will use State pattern to implement above TV Remote example.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

import javax.naming.Context;

/\*

\* we will create State interface that will define the method that should be implemented \* by different

\* concrete states and context class

\*/

interface State{

public void doAction();

}

/\*

\* State Design pattern concrete State Implementations

\* In our example ,we can have two states - one for turning TV on and another to turn it \* OFF. so we will create

\* two concrete state implementations for these behaviors

\*/

class TVStartState implements State{

@Override

public void doAction() {

System.out.println("TV is turned on");

}

}

class TVStopState implements State{

@Override

public void doAction() {

System.out.println("TV is turned off");

}

}

/\*

\* now we are ready to implement our context object that will change its behavior based \* on its internal state

\* State design pattern context implementation

\*/

class TVContext implements State{

private State tvState;

public void setState(State state) {

this.tvState = state;

}

public State getState() {

return tvState;

}

@Override

public void doAction() {

this.tvState.doAction();

}

}

/\*

\* Notice that Context also implements State and keep a reference of its current state \* and forwards the request

\* to the state implementation

\*/

/\*

\* Testing state pattern

\*/

public class TestStatePattern {

public static void main(String[] args) {

TVContext tvContext = new TVContext();

State tvStartState = new TVStartState();

State tvStopState = new TVStopState();

tvContext.setState(tvStartState);

tvContext.doAction();

tvContext.setState(tvStopState);

tvContext.doAction();

}

}