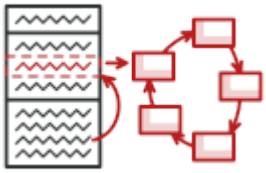


State in Java

State is a behavioral design pattern that allows an object to change the behavior when its internal state changes.

The pattern extracts state-related behaviors into separate state classes and forces the original object to delegate the work to an instance of these classes, instead of acting on its own.

Learn more about State →

Navigation

Complexity: ★☆☆

Popularity: ★★☆

Usage examples: The State pattern is commonly used in Java to convert massive `switch`-base state machines into objects.

`javax.faces.lifecycle.LifeCycle#execute()` (controlled by the `FacesServlet` : behavior is dependent on current phase (state) of JSF lifecycle)

Identification: The State pattern can be recognized by methods that change their behavior depending on the objects' state. You can confirm identification if you see that this state can be controlled or replaced by other objects, including state objects themselves.

Interface of a media player

In this example, the State pattern lets the same media player controls behave differently, depending on the current playback state. The main class of the player contains a reference to a state object, which performs most of the work for the player. Some actions may end-up replacing the state object with another, which changes the way the player reacts to the user interactions.

states

states/State.java: Common state interface

```
package refactoring_guru.state.example.states;

import refactoring_guru.state.example.ui.Player;

/**
 * Common interface for all states.
 */
public abstract class State {
    Player player;

    /**
     * Context passes itself through the state constructor. This may help a
     * state to fetch some useful context data if needed.
     */
    State(Player player) {
        this.player = player;
    }

    public abstract String onLock();
    public abstract String onPlay();
    public abstract String onNext();
    public abstract String onPrevious();
}
```

states/LockedState.java

```
import refactoring_guru.state.example.ui.Player;

/**
 * Concrete states provide the special implementation for all interface methods.
 */
public class LockedState extends State {

    LockedState(Player player) {
        super(player);
        player.setPlaying(false);
    }

    @Override
    public String onLock() {
        if (player.isPlaying()) {
            player.changeState(new ReadyState(player));
            return "Stop playing";
        } else {
            return "Locked...";
        }
    }

    @Override
    public String onPlay() {
        player.changeState(new ReadyState(player));
        return "Ready";
    }

    @Override
    public String onNext() {
        return "Locked...";
    }

    @Override
    public String onPrevious() {
        return "Locked...";
    }
}
```

states/ReadyState.java

```
package refactoring_guru.state.example.states;

import refactoring_guru.state.example.ui.Player;

/**
 * They can also trigger state transitions in the context.
 */
public class ReadyState extends State {
```

```
    }

    @Override
    public String onLock() {
        player.changeState(new LockedState(player));
        return "Locked...";
    }

    @Override
    public String onPlay() {
        String action = player.startPlayback();
        player.changeState(new PlayingState(player));
        return action;
    }

    @Override
    public String onNext() {
        return "Locked...";
    }

    @Override
    public String onPrevious() {
        return "Locked...";
    }
}
```

states/PlayingState.java

```
package refactoring_guru.state.example.states;

import refactoring_guru.state.example.ui.Player;

public class PlayingState extends State {

    PlayingState(Player player) {
        super(player);
    }

    @Override
    public String onLock() {
        player.changeState(new LockedState(player));
        player.setCurrentTrackAfterStop();
        return "Stop playing";
    }

    @Override
    public String onPlay() {
        player.changeState(new ReadyState(player));
        return "Paused...";
    }
```

```
    public String onNext() {
        return player.nextTrack();
    }

    @Override
    public String onPrevious() {
        return player.previousTrack();
    }
}
```

ui

ui/Player.java: Player primary code

```
package refactoring_guru.state.example.ui;

import refactoring_guru.state.example.states.ReadyState;
import refactoring_guru.state.example.states.State;

import java.util.ArrayList;
import java.util.List;

public class Player {
    private State state;
    private boolean playing = false;
    private List<String> playlist = new ArrayList<>();
    private int currentTrack = 0;

    public Player() {
        this.state = new ReadyState(this);
        setPlaying(true);
        for (int i = 1; i <= 12; i++) {
            playlist.add("Track " + i);
        }
    }

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

    public State getState() {
        return state;
    }

    public void setPlaying(boolean playing) {
        this.playing = playing;
    }

    public boolean isPlaying() {
        return playing;
```

```
    public String startPlayback() {
        return "Playing " + playlist.get(currentTrack);
    }

    public String nextTrack() {
        currentTrack++;
        if (currentTrack > playlist.size() - 1) {
            currentTrack = 0;
        }
        return "Playing " + playlist.get(currentTrack);
    }

    public String previousTrack() {
        currentTrack--;
        if (currentTrack < 0) {
            currentTrack = playlist.size() - 1;
        }
        return "Playing " + playlist.get(currentTrack);
    }

    public void setCurrentTrackAfterStop() {
        this.currentTrack = 0;
    }
}
```

ui/UI.java: Player's GUI

```
package refactoring_guru.state.example.ui;

import javax.swing.*;
import java.awt.*;

public class UI {
    private Player player;
    private static JTextField textField = new JTextField();

    public UI(Player player) {
        this.player = player;
    }

    public void init() {
        JFrame frame = new JFrame("Test player");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JPanel context = new JPanel();
        context.setLayout(new BoxLayout(context, BoxLayout.Y_AXIS));
        frame.getContentPane().add(context);
        JPanel buttons = new JPanel(new FlowLayout(FlowLayout.CENTER));
        context.add(textField);
        context.add(buttons);
```

```
        // states can handle the input differently.
        JButton play = new JButton("Play");
        play.addActionListener(e -> textField.setText(player.getState().onPlay()));
        JButton stop = new JButton("Stop");
        stop.addActionListener(e -> textField.setText(player.getState().onLock()));
        JButton next = new JButton("Next");
        next.addActionListener(e -> textField.setText(player.getState().onNext()));
        JButton prev = new JButton("Prev");
        prev.addActionListener(e -> textField.setText(player.getState().onPrevious()));
        frame.setVisible(true);
        frame.setSize(300, 100);
        buttons.add(play);
        buttons.add(stop);
        buttons.add(next);
        buttons.add(prev);
    }
}
```

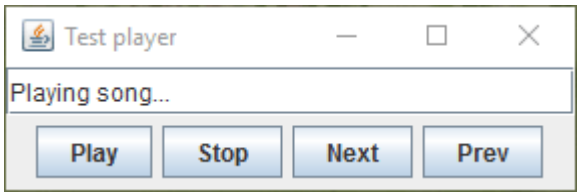
Demo.java: Initialization code

```
package refactoring_guru.state.example;

import refactoring_guru.state.example.ui.Player;
import refactoring_guru.state.example.ui.UI;

/**
 * Demo class. Everything comes together here.
 */
public class Demo {
    public static void main(String[] args) {
        Player player = new Player();
        UI ui = new UI(player);
        ui.init();
    }
}
```

OutputDemo.png: Screenshot



RETURN

READ NEXT