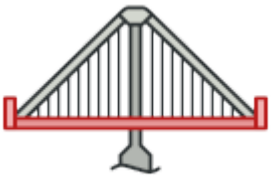

Bridge in Java

Bridge is a structural design pattern that divides business logic or huge class into separate class hierarchies that can be developed independently.

One of these hierarchies (often called the Abstraction) will get a reference to an object of the second hierarchy (Implementation). The abstraction will be able to delegate some (sometimes, most) of its calls to the implementations object. Since all implementations will have a common interface, they'd be interchangeable inside the abstraction.

Learn more about Bridge →

Navigation

Intro

Bridge between devices and remote controls
devices
Device
Radio
Tv
remotes
Remote
BasicRemote
AdvancedRemote
Demo
OutputDemo

Complexity: ★★★

Usage examples: The Bridge pattern is especially useful when dealing with cross-platform apps, supporting multiple types of database servers or working with several API providers of a certain kind (for example, cloud platforms, social networks, etc.)

Identification: Bridge can be recognized by a clear distinction between some controlling entity and several different platforms that it relies on.

Bridge between devices and remote controls

This example shows separation between the classes of remotes and devices that they control.

Remotes act as abstractions, and devices are their implementations. Thanks to the common interfaces, the same remotes can work with different devices and vice versa.

The Bridge pattern allows changing or even creating new classes without touching the code of the opposite hierarchy.

devices

devices/Device.java: Common interface of all devices

```
package refactoring_guru.bridge.example.devices;

public interface Device {
    boolean isEnabled();

    void enable();

    void disable();

    int getVolume();

    void setVolume(int percent);

    int getChannel();

    void setChannel(int channel);

    void printStatus();
}
```

```
package refactoring_guru.bridge.example.devices;

public class Radio implements Device {
    private boolean on = false;
    private int volume = 30;
    private int channel = 1;

    @Override
    public boolean isEnabled() {
        return on;
    }

    @Override
    public void enable() {
        on = true;
    }

    @Override
    public void disable() {
        on = false;
    }

    @Override
    public int getVolume() {
        return volume;
    }

    @Override
    public void setVolume(int volume) {
        if (volume > 100) {
            this.volume = 100;
        } else if (volume < 0) {
            this.volume = 0;
        } else {
            this.volume = volume;
        }
    }

    @Override
    public int getChannel() {
        return channel;
    }

    @Override
    public void setChannel(int channel) {
        this.channel = channel;
    }

    @Override
```

```
        System.out.println("| I'm radio.");
        System.out.println("| I'm " + (on ? "enabled" : "disabled"));
        System.out.println("| Current volume is " + volume + "%");
        System.out.println("| Current channel is " + channel);
        System.out.println("----------------------------------\n");
    }
}
```

devices/Tv.java: TV

```
package refactoring_guru.bridge.example.devices;

public class Tv implements Device {
    private boolean on = false;
    private int volume = 30;
    private int channel = 1;

    @Override
    public boolean isEnabled() {
        return on;
    }

    @Override
    public void enable() {
        on = true;
    }

    @Override
    public void disable() {
        on = false;
    }

    @Override
    public int getVolume() {
        return volume;
    }

    @Override
    public void setVolume(int volume) {
        if (volume > 100) {
            this.volume = 100;
        } else if (volume < 0) {
            this.volume = 0;
        } else {
            this.volume = volume;
        }
    }
```

```
    public int getChannel() {
        return channel;
    }

    @Override
    public void setChannel(int channel) {
        this.channel = channel;
    }

    @Override
    public void printStatus() {
        System.out.println("-------------------------------------");
        System.out.println("| I'm TV set.");
        System.out.println("| I'm " + (on ? "enabled" : "disabled"));
        System.out.println("| Current volume is " + volume + "%");
        System.out.println("| Current channel is " + channel);
        System.out.println("-------------------------------------\n");
    }
}
```

remotes

remotes/Remote.java: Common interface for all remotes

```
package refactoring_guru.bridge.example.remotes;

public interface Remote {
    void power();

    void volumeDown();

    void volumeUp();

    void channelDown();

    void channelUp();
}
```

remotes/BasicRemote.java: Basic remote control

```
package refactoring_guru.bridge.example.remotes;

import refactoring_guru.bridge.example.devices.Device;
```

```
    protected Device device;

    public BasicRemote() {}

    public BasicRemote(Device device) {
        this.device = device;
    }

    @Override
    public void power() {
        System.out.println("Remote: power toggle");
        if (device.isEnabled()) {
            device.disable();
        } else {
            device.enable();
        }
    }

    @Override
    public void volumeDown() {
        System.out.println("Remote: volume down");
        device.setVolume(device.getVolume() - 10);
    }

    @Override
    public void volumeUp() {
        System.out.println("Remote: volume up");
        device.setVolume(device.getVolume() + 10);
    }

    @Override
    public void channelDown() {
        System.out.println("Remote: channel down");
        device.setChannel(device.getChannel() - 1);
    }

    @Override
    public void channelUp() {
        System.out.println("Remote: channel up");
        device.setChannel(device.getChannel() + 1);
    }
}
```

remotes/AdvancedRemote.java: Advanced remote control

```
package refactoring_guru.bridge.example.remotes;
```

```
public class AdvancedRemote extends BasicRemote {

    public AdvancedRemote(Device device) {
        super.device = device;
    }

    public void mute() {
        System.out.println("Remote: mute");
        device.setVolume(0);
    }
}
```

Demo.java: Client code

```
package refactoring_guru.bridge.example;

import refactoring_guru.bridge.example.devices.Device;
import refactoring_guru.bridge.example.devices.Radio;
import refactoring_guru.bridge.example.devices.Tv;
import refactoring_guru.bridge.example.remotes.AdvancedRemote;
import refactoring_guru.bridge.example.remotes.BasicRemote;

public class Demo {
    public static void main(String[] args) {
        testDevice(new Tv());
        testDevice(new Radio());
    }

    public static void testDevice(Device device) {
        System.out.println("Tests with basic remote.");
        BasicRemote basicRemote = new BasicRemote(device);
        basicRemote.power();
        device.printStatus();

        System.out.println("Tests with advanced remote.");
        AdvancedRemote advancedRemote = new AdvancedRemote(device);
        advancedRemote.power();
        advancedRemote.mute();
        device.printStatus();
    }
}
```

OutputDemo.txt: Execution result

```
Remote: power toggle
-------------------------------------
| I'm TV set.
| I'm enabled
| Current volume is 30%
| Current channel is 1
-------------------------------------

Tests with advanced remote.
Remote: power toggle
Remote: mute
-------------------------------------
| I'm TV set.
| I'm disabled
| Current volume is 0%
| Current channel is 1
-------------------------------------

Tests with basic remote.
Remote: power toggle
-------------------------------------
| I'm radio.
| I'm enabled
| Current volume is 30%
| Current channel is 1
-------------------------------------

Tests with advanced remote.
Remote: power toggle
Remote: mute
-------------------------------------
| I'm radio.
| I'm disabled
| Current volume is 0%
| Current channel is 1
-------------------------------------
```

READ NEXT

Home
Refactoring
Design Patterns
Premium Content
Forum