

به نام خداوند بخشنده مهربان

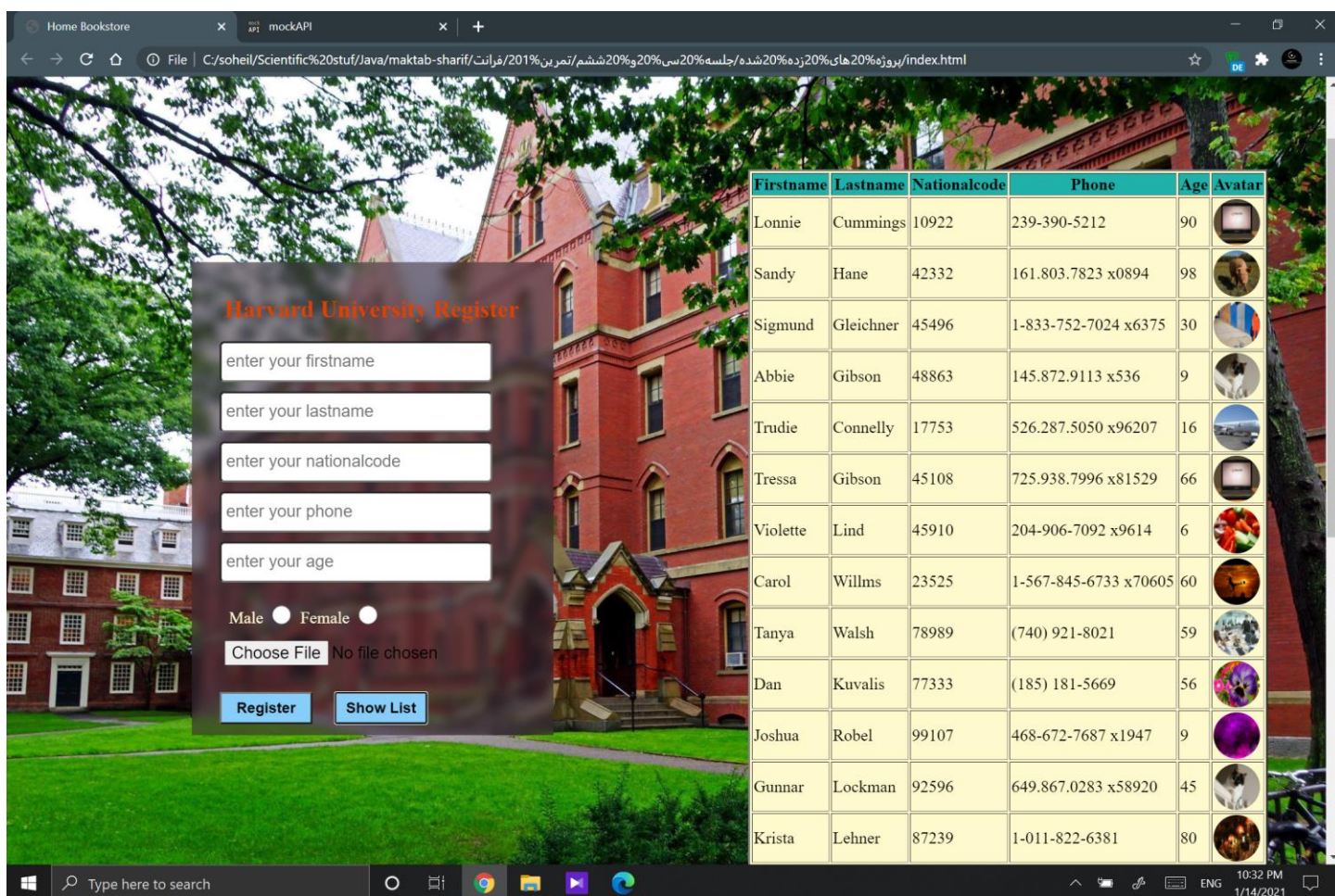
محمدرضا مسیب زاده

تمرین جاوا :

[https://github.com/M-Mosaiebzadeh/Maktab\\_W18.git](https://github.com/M-Mosaiebzadeh/Maktab_W18.git)

تمرین شماره ۱:

اجرای تمرین :



**Harvard University Register**

enter your firstname  
enter your lastname  
enter your nationalcode  
enter your phone  
enter your age

Male ☐ Female ☐

Choose File No file chosen

Register Show List

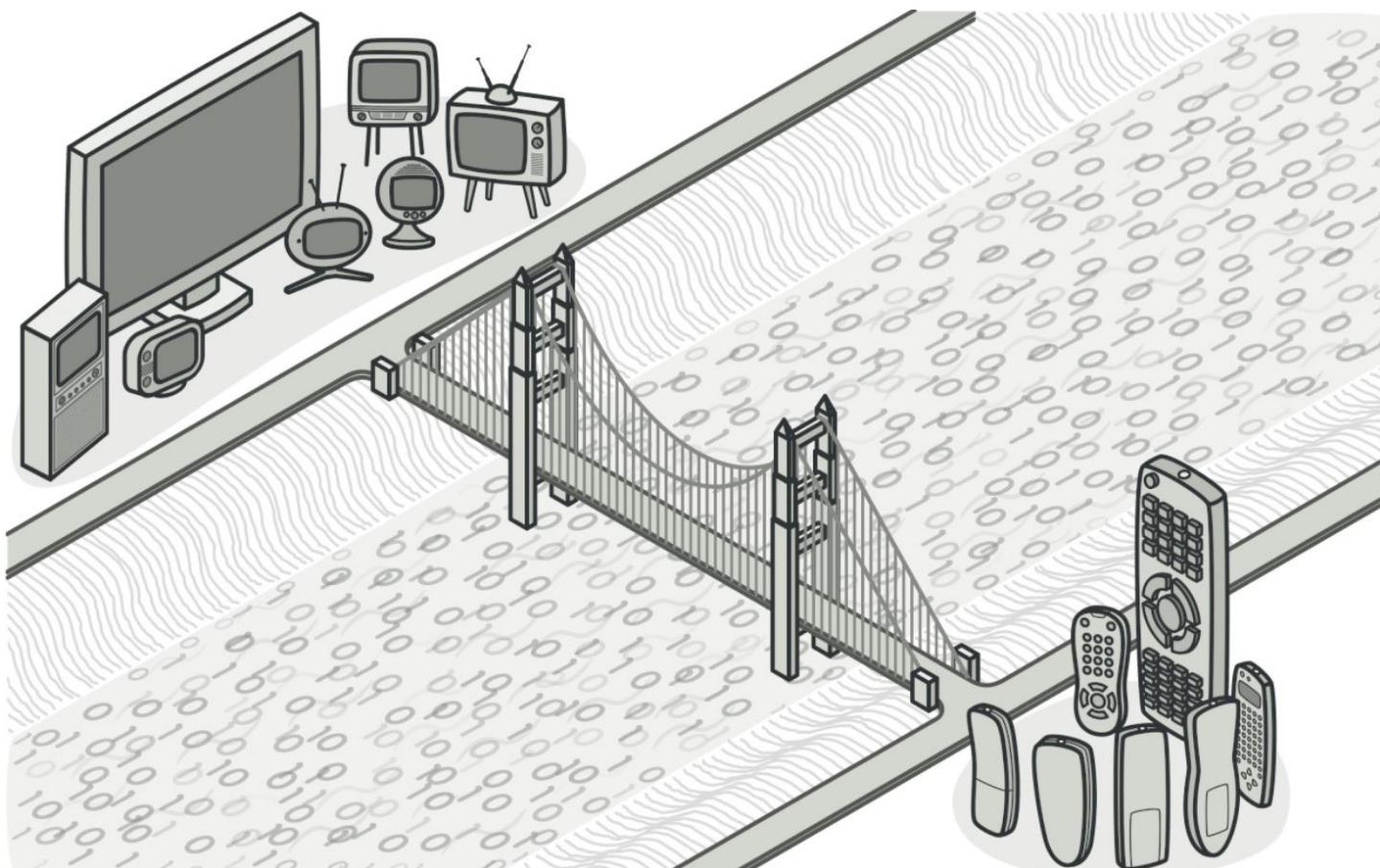
Firstname	Lastname	Nationalcode	Phone	Age	Avatar
Lonnie	Cummings	10922	239-390-5212	90	
Sandy	Hane	42332	161.803.7823 x0894	98	
Sigmund	Gleichner	45496	1-833-752-7024 x6375	30	
Abbie	Gibson	48863	145.872.9113 x536	9	
Trudie	Connelly	17753	526.287.5050 x96207	16	
Tressa	Gibson	45108	725.938.7996 x81529	66	
Violette	Lind	45910	204-906-7092 x9614	6	
Carol	Willms	23525	1-567-845-6733 x70605	60	
Tanya	Walsh	78989	(740) 921-8021	59	
Dan	Kuvalis	77333	(185) 181-5669	56	
Joshua	Robel	99107	468-672-7687 x1947	9	
Gunnar	Lockman	92596	649.867.0283 x58920	45	
Krista	Lehner	87239	1-011-822-6381	80	

## Bridge Design Pattern

Bridge is a **structural** design pattern that lets you **split** a **large class** or a **set of closely related classes** into **two separate hierarchies**—abstraction and implementation—which can be developed independently of each other.

دیزاین پترن ساختاری است.

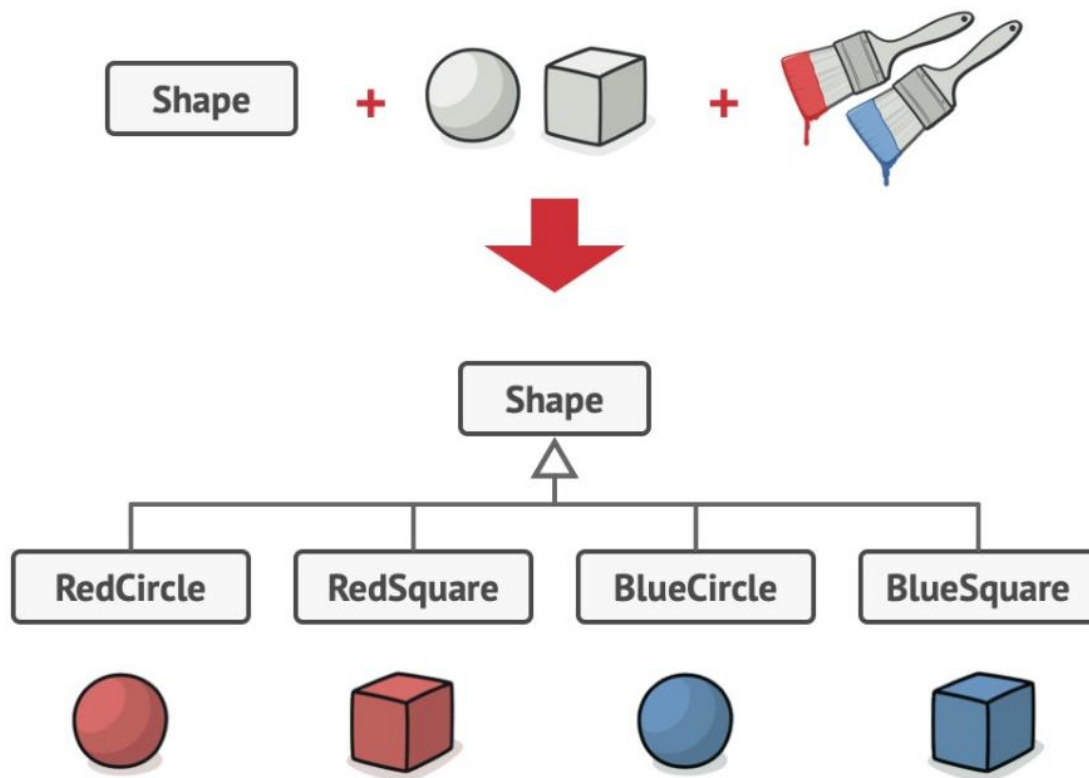
به ما این اجازه رو میده که یک کلاس بزرگ یا مجموعه ای از کلاس های وابسته به هم رو به دو سلسله مراتب ابسترکشن و پیاده سازی تبدیل کنیم که بتونیم مستقل از هم توسعه شون بدیم.



Say you have a **geometric Shape class** with a pair of subclasses: **Circle and Square**. You want to extend this class hierarchy to **incorporate colors**, so you plan to create **Red and Blue** shape subclasses. However, **since you already have two subclasses, you'll need to create four class combinations** such as BlueCircle and RedSquare.

فرض کنیم یک کلاس اشکال داریم که کلاس دایره و مربع اکستندش کردن. حالا میخوایم سلسه مراتب این کلاس رو گسترش بدیم و با رنگ ترکیب کنیم. فرض کنیم دو رنگ قرمز و آبی داریم.

ما میخوایم اشکالی با رنگ های آبی و قرمز بسازیم، اگر چه دو ساب کلاس رنگ داریم ولی باید چهارتا کلاس دایره آبی، دایره قرمز، مربع آبی و مربع قرمز را داشته باشیم.



*Number of class combinations grows in geometric progression.*

**Adding new shape types and colors** to the hierarchy **will grow it exponentially**. For example, to add a triangle shape you'd need to introduce two subclasses, one for each color. And after

that, adding a new color would require creating three subclasses, one for each shape type. The further we go, the worse it becomes.

با اضافه کردن شکل یا رنگ جدید باعث میشود تعداد کلاس های ما به صورت اکسپوننشیالی افزایش یابد.  
مثلا اگر مثلث را به شکل ها اضافه کنیم نیاز به ساخت دو ساب کلاس مثلث آبی و مثلث قرمز داریم.  
یا اگر یک رنگ دیگر به این کلاس اضافه کنیم نیاز به ساخت سه ساب کلاس دیگر داریم.

## Solution

This problem occurs because we're trying to **extend the shape classes in two independent dimensions**: by form and by color. That's a very common issue with class inheritance.

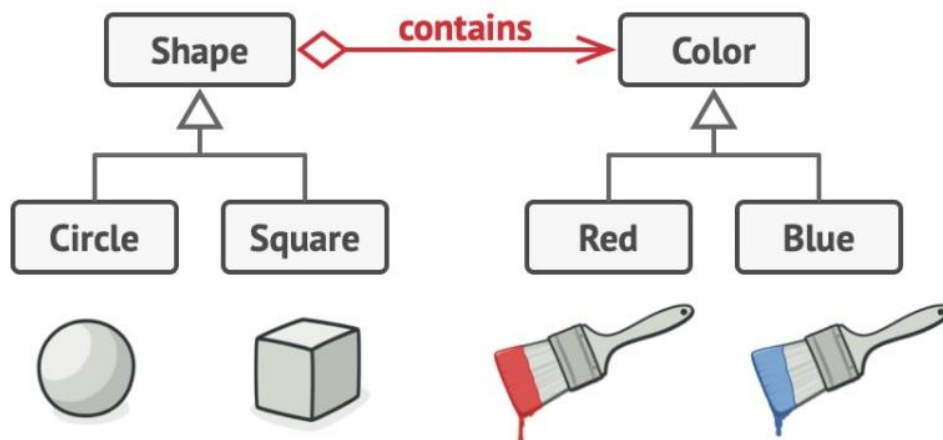
**The Bridge pattern attempts to solve this problem by switching from inheritance to the object composition.** What this means is that you extract one of the dimensions into a separate class hierarchy, so that the original classes will reference an object of the new hierarchy, instead of having all of its state and behaviors within one class.

این مشکل زمانی اتفاق میفته که ما تلاش میکنیم کلاس اشیا رو از دو بعد مستقل گسترش بدیم.

دیزاین پترن بریج با تغییر دادن از ارث بری به ترکیب شی، این مشکل رو حل کرده.

به این معنی که یکی از کلاس ها رو درون کلاس دیگری میذاریم با این کار کلاس جدیدی که میسازیم ارجاع داره به یک شی از کلاس دیگه به جای این که تمام فیلد ها و متد هاش رو دارا باشه.





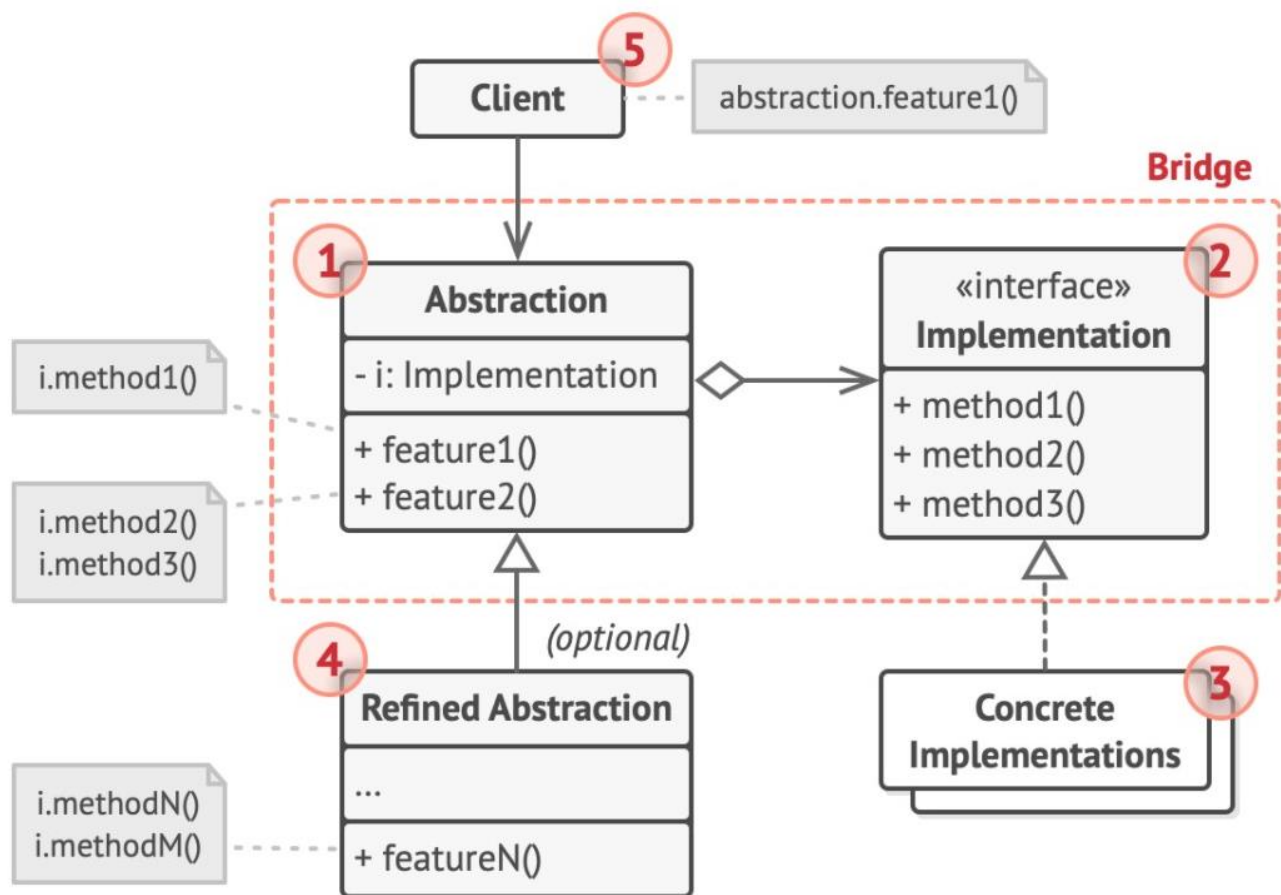
*You can prevent the explosion of a class hierarchy by transforming it into several related hierarchies.*

Following this approach, we can extract the color-related code into its own class with two subclasses: Red and Blue. The Shape class then gets a reference field pointing to one of the color objects. Now the shape can delegate any color-related work to the linked color object. That reference will act as a bridge between the Shape and Color classes. From now on, adding new colors won't require changing the shape hierarchy, and vice versa.

با این رویکرد ما میتونیم کد مربوط به هر رنگ رو در کلاس خودش بذاریم با دو ساب کلاس قرمز و آبی.

کلاس شکل ما یک ارجاع از ابجکتی از کلاس رنگ رو در خودش نگه میداره. در حقیقت این ارجاع یک پل بین کلاس های شکل و رنگ است. با این کار دیگر با اضافه کردن یک رنگ دیگر نیازی نیست به تعداد کلاس های شکل، از آن ساب کلاس بسازیم.

## Structure



- 1- The Abstraction provides high-level control logic. It relies on the implementation object to do the actual low-level work.

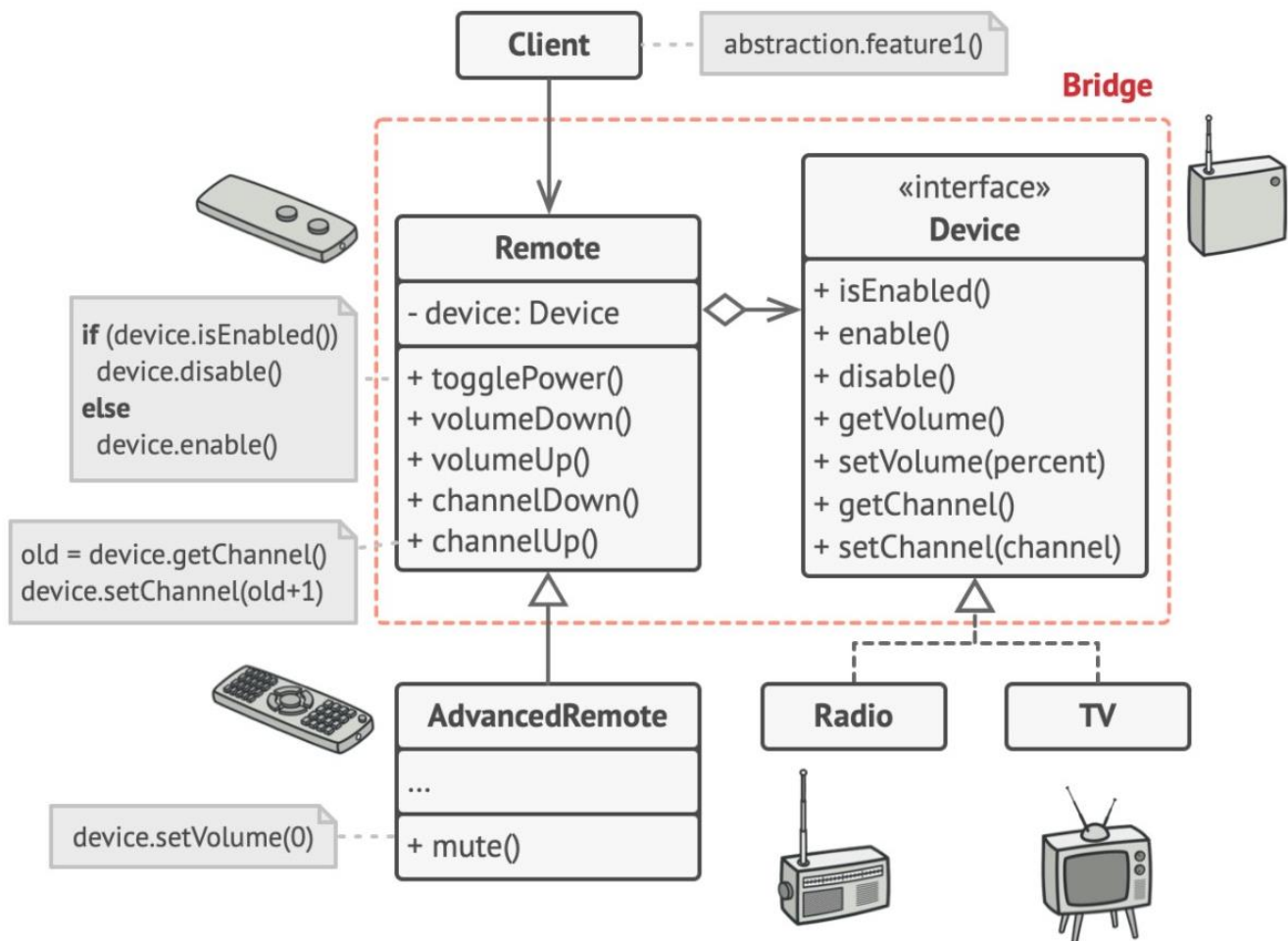
ابسترکشن منطق کنترل سطح بالا رو فراهم میکنه و براساس شی پیاده سازی شده کار واقعی سطح پایین رو انجام میده.

- 2- The Implementation declares the interface that's common for all concrete implementations. An abstraction can only communicate with an implementation object via methods that are declared here.

یک سری پیاده سازی از اینترفیس میسازیم. ابسترکشن ما میتونه با شی اینترفیس ما از طریق متدهاش ارتباط برقرار کنه.

3- The abstraction may list the same methods as the implementation, but usually the abstraction declares some complex behaviors that rely on a wide variety of primitive operations declared by the implementation.

ابسترکشن ما ممکنه لیست متد های یکسانی با ایمپلیمنتیشن ما داشته باشه، اما به طور معمول بیان میکند برخی از رفتار های پیچیده (متدش) رو که وابسته به عملکردهای اولیه (متدهای) که توسط ایمپلیمنتیشن بیان شده اند.



*The original class hierarchy is divided into two parts: devices and remote controls.*

در این طراحی دو وسیله تلویزیون و رادیو را داریم و دو کنترل ساده و پیشرفته را داریم، با استفاده از پترن پل دیگر نیازی نیست برای هر دو وسیله دو کنترل بسازیم (چهار کلاس) و فقط کنترل میسازیم که هر کدام یک شی از وسیله را در خود نگه میدارند، این کار باعث رعایت کردن دو اصل **solid** میشود:

- 1- Open/Closed Principle. You can introduce new abstractions and implementations independently from each other.
- 2- Single Responsibility Principle. You can focus on high-level logic in the abstraction and on platform details in the implementation.



```

1 package ir.maktab.structural.bridge.device;
2
3 public interface Device {
4     boolean isOn();
5     void on();
6     void off();
7     int getVolume();
8     void setVolume(int volume);
9     int getChannel();
10    void setChannel(int channel);
11 }
12
13

```

کلاس Tv که آن را پیاده سازی کرده اند:

```

1 package ir.maktab.structural.bridge.device;
2
3 public class Tv implements Device {
4     private boolean on = false;
5     private int volume = 50;
6     private int channel = 1;
7     @Override
8     public boolean isOn() {
9         return on;
10    }
11
12    @Override
13    public void on() {
14        this.on = true;
15    }
16
17    @Override
18    public void off() {
19        this.on = false;
20    }
21
22    @Override
23    public int getVolume() {
24        return volume;
25    }
26
27    @Override
28    public void setVolume(int volume) {
29        if (volume > 100)
30            this.volume = 100;

```

```

28 public void setVolume(int volume) {
29     if (volume > 100)
30         this.volume = 100;
31     else if (volume < 0)
32         this.volume = 0;
33     else
34         this.volume = volume;
35 }
36
37 @Override
38 public int getChannel() {
39     return channel;
40 }
41
42 @Override
43 public void setChannel(int channel) {
44     if (channel > 20 || channel == 0)
45         this.channel = 1;
46     else
47         this.channel = channel;
48 }
49
50 @Override
51 public String toString() {
52     return "Tv{" +
53         "on=" + on +
54         ", volume=" + volume +
55         ", channel=" + channel +
56         '}';
57 }

```

کلاس Radio که آن را پیاده سازی کرده است:

```

1 package ir.maktab.structural.bridge.device;
2
3 public class Radio implements Device {
4     private boolean on = false;
5     private int volume = 50;
6     private int channel = 1;
7     @Override
8     public boolean isOn() {
9         return on;
10    }
11
12    @Override
13    public void on() {
14        this.on = true;
15    }
16
17    @Override
18    public void off() {
19        this.on = false;
20    }
21
22    @Override
23    public int getVolume() {
24        return volume;
25    }
26
27    @Override
28    public void setVolume(int volume) {
29        if (volume > 100)
30            this.volume = 100;
31        else if (volume < 0)
32            this.volume = 0;
33        else
34            this.volume = volume;
35    }
36 }

```

```
Radio.java ×
28 public void setVolume(int volume) {
29     if (volume > 100)
30         this.volume = 100;
31     else if (volume < 0)
32         this.volume = 0;
33     else
34         this.volume = volume;
35 }
36
37 @Override
38 public int getChannel() {
39     return channel;
40 }
41
42 @Override
43 public void setChannel(int channel) {
44     if (channel > 100 || channel == 0)
45         this.channel = 1;
46     else
47         this.channel = channel;
48 }
49
50 @Override
51 public String toString() {
52     return "Radio{" +
53         "on=" + on +
54         ", volume=" + volume +
55         ", channel=" + channel +
56         '}';
57 }
```

ابستركت كلاس: Remote

```
Radio.java × Remote.java × BasicRemote.java ×
1 package ir.maktab.structural.bridge.remote;
2
3 import ir.maktab.structural.bridge.device.Device;
4
5 public abstract class Remote {
6     protected Device device;
7
8     public Remote(Device device) { this.device = device; }
9
10    public abstract void power();
11
12
13    public abstract void volumeUp();
14
15    public abstract void volumeDown();
16
17    public abstract void channelUp();
18
19    public abstract void channelDown();
20 }
21
```

کلاس BasicRemote که ابسترکت کلاس Remote را پیاده سازی کرده است:

```
BasicRemote.java x AdvancedRemote.java x
1 package ir.maktab.structural.bridge.remote;
2
3 import ir.maktab.structural.bridge.device.Device;
4
5 public class BasicRemote extends Remote{
6
7     public BasicRemote(Device device) {
8         super(device);
9     }
10
11     public void power() {
12         if (device.isOn()){
13             device.off();
14             System.out.println("turn off");
15         }else {
16             device.on();
17             System.out.println("turn on");
18         }
19     }
20
21     public void volumeUp() {
22         int oldVolume = device.getVolume();
23         device.setVolume(oldVolume + 1);
24         System.out.println(oldVolume + " => " + device.getVolume());
25     }
26
27     public void volumeDown() {
28         int oldVolume = device.getVolume();
29         device.setVolume(oldVolume - 1);
30         System.out.println(oldVolume + " => " + device.getVolume());
31
32
33     public void channelUp() {
34         int oldChannel = device.getChannel();
35         device.setChannel(oldChannel + 1);
36         System.out.println(oldChannel + " => " + device.getChannel());
37     }
38
39     public void channelDown() {
40         int oldChannel = device.getChannel();
41         device.setChannel(oldChannel - 1);
42         System.out.println(oldChannel + " => " + device.getChannel());
43     }
44
45 }
```

کلاس AdvancedRemote که کلاس BasicRemote را پیاده کرده است و تنها به آن یک قابلیت میوت اضافه کرده است:

```
AdvancedRemote.java x
1 package ir.maktab.structural.bridge.remote;
2
3 import ir.maktab.structural.bridge.device.Device;
4
5 public class AdvancedRemote extends BasicRemote{
6
7
8     public AdvancedRemote(Device device) {
9         super(device);
10    }
11
12    public void mute() {
13        device.setVolume(0);
14        System.out.println("sound is muted");
15    }
16 }
17
```

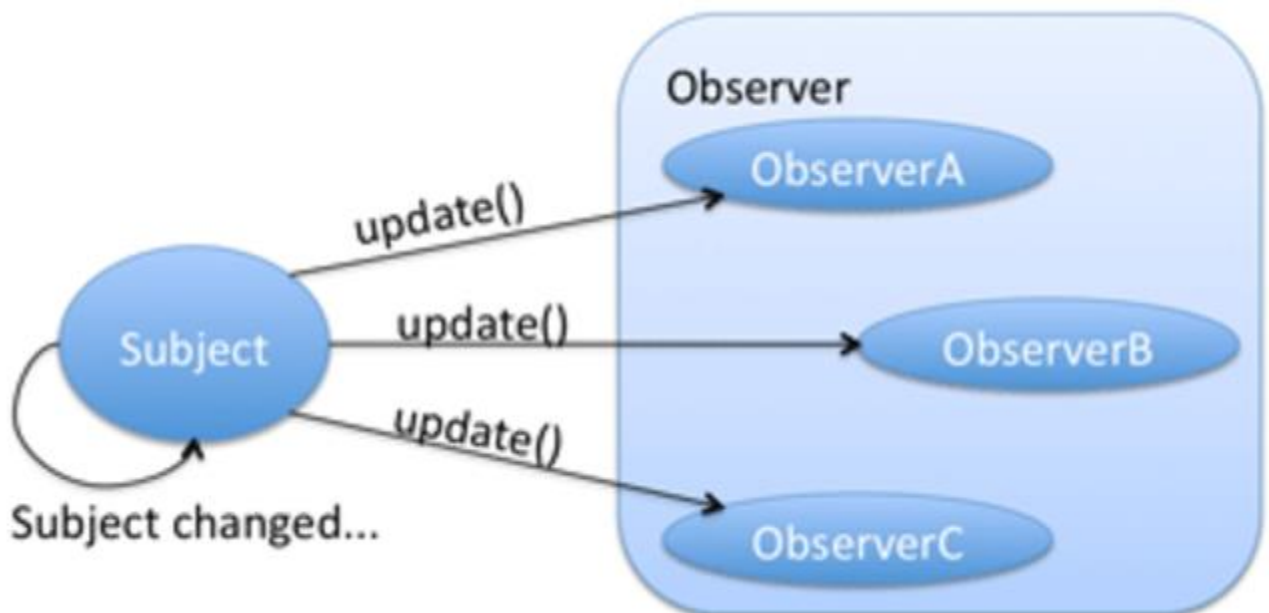
کلاس BridgeDemo:

```
AdvancedRemote.java x bridgeDemo.java x
4 import ir.maktab.structural.bridge.device.Radio;
5 import ir.maktab.structural.bridge.device.Tv;
6 import ir.maktab.structural.bridge.remote.AdvancedRemote;
7 import ir.maktab.structural.bridge.remote.BasicRemote;
8
9 public class bridgeDemo {
10     public static void main(String[] args) {
11         Device tv = new Tv();
12         Device radio = new Radio();
13         System.out.println("=====");
14
15         AdvancedRemote advTvRemote = new AdvancedRemote(tv);
16         System.out.println(tv);
17
18         advTvRemote.power();
19         advTvRemote.channelDown();
20         advTvRemote.mute();
21
22         System.out.println(tv);
23
24         System.out.println("=====");
25
26         BasicRemote basicRadioRemote = new BasicRemote(radio);
27         System.out.println(radio);
28
29         basicRadioRemote.power();
30         for (int i = 0; i < 10; i++) {
31             basicRadioRemote.channelUp();
32         }
33         System.out.println(radio);
34     }
35 }
```

```
"C:\Program Files\Java\jdk1.8.0_261\bin\java.exe" ...
=====
Tv{on=false, volume=50, channel=1}
turn on
1 => 1
sound is muted
Tv{on=true, volume=0, channel=1}
=====
Radio{on=false, volume=50, channel=1}
turn on
1 => 2
2 => 3
3 => 4
4 => 5
5 => 6
6 => 7
7 => 8
8 => 9
9 => 10
10 => 11
Radio{on=true, volume=50, channel=11}
```

## Observer

این الگو زیرمجموعه ی الگوهای رفتاری (Behavioral) هست.





الگوی ناظر یا همان Observer یک الگوی طراحی نرم افزار است که در آن یک شی به نام موضوع (subject)، فهرست وابستگی هایش را با نام ناظران (observers) نگه می دارد و هرگونه تغییر در وضعیتش را به طور خودکار و معمولاً با صدا کردن یکی از روش های آن به اطلاع آن اشیا می رساند.

اول به کلاس abstract به اسم Observer بصورت زیر تعریف میکنیم:

این ابسترکت کلاس متدی داره به اسم update داره که abstract هست و طبیعتاً هرکلاسی از این کلاس ارثیری کنه باید این تابع رو پیاده سازی بکنه. و یه شی هم از کلاس Subject نگه میداره

```
1 package ir.maktab.behavioral.observer;
2
3 public abstract class Observer {
4
5     private Subject subject;
6
7     public abstract void update();
8
9 }
10
```

بعد کلاس Subject رو تعریف میکنیم:

این کلاس لیستی از Observer ها داره. تابعی با اسم add که به Observer به لیست اضافه میکنه

تابع getState و setState که توابع setter و getter پارامتر state هستن

در نهایت هم تابع execute که تابع update تمامی observerهای موجود توی لیست رو صدا میزنه

```

1  package ir.maktab.behavioral.observer;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  public class Subject {
7      private List<Observer> observers = new ArrayList<>();
8      private int state;
9
10     public void add(Observer observer) {
11         observers.add(observer);
12     }
13
14     public int getState() {
15         return state;
16     }
17
18     private void execute() {
19         observers.forEach(observer -> observer.update());
20     }
21
22     public void setState(int state) {
23         this.state = state;
24         execute();
25     }
26 }
27

```

حالا سه تا کلاس که Observer را پیاده سازی کرده اند، تعریف میکنیم:

کلاس HexObserver (تبدیل مبنای ده دهی به هگزادسیمال (Hexadecimal))

کلاس OctObserver (تبدیل مبنای ده دهی به مبنای هشت هشتی (Octal))

و در نهایت کلاس BinObserver (تبدیل مبنای ده دهی به باینری)

```

BinObserver.java x ObserverDemo.java x
1 package ir.maktab.behavioral.observer;
2
3 public class BinObserver extends Observer{
4
5     public BinObserver(Subject subject) {
6         this.subject = subject;
7         this.subject.add(this);
8     }
9
10    @Override
11    public void update() {
12        System.out.println(subject.getState() + " => to Bin: " + Integer.toBinaryString(subject.getState()));
13    }
14 }
15

```

```

ObserverDemo.java x OctObserver.java x
1 package ir.maktab.behavioral.observer;
2
3 public class OctObserver extends Observer{
4
5     public OctObserver(Subject subject) {
6         this.subject = subject;
7         this.subject.add(this);
8     }
9
10    @Override
11    public void update() {
12        System.out.println(subject.getState() + " => to Oct: " + Integer.toOctalString(subject.getState()));
13    }
14 }
15

```

```

ObserverDemo.java x HexObserver.java x
1 package ir.maktab.behavioral.observer;
2
3 public class HexObserver extends Observer {
4
5     public HexObserver(Subject subject) {
6         this.subject = subject;
7         this.subject.add(this);
8     }
9
10    @Override
11    public void update() {
12        System.out.println(subject.getState() + " => to Hex: " + Integer.toHexString(subject.getState()));
13    }
14 }
15

```

```

1 package ir.maktab.behavioral.observer;
2
3 public class ObserverDemo {
4     public static void main(String[] args) {
5         Subject subject = new Subject();
6         Observer binObserver = new BinObserver(subject);
7         Observer octObserver = new OctObserver(subject);
8         Observer hexObserver = new HexObserver(subject);
9
10        subject.setState(12);
11        System.out.println("=====");
12        subject.setState(193);
13    }
14 }
15

```

خروجی برنامه:

```

Run: ObserverDemo x
"C:\Program Files\Java\jdk1.8.0_261\bin\java.exe" ...
12 => to Bin: 1100
12 => to Oct: 14
12 => to Hex: c
=====
193 => to Bin: 11000001
193 => to Oct: 301
193 => to Hex: c1

Process finished with exit code 0

```