

Observer Design Pattern

Notes by Bhavuk Jain

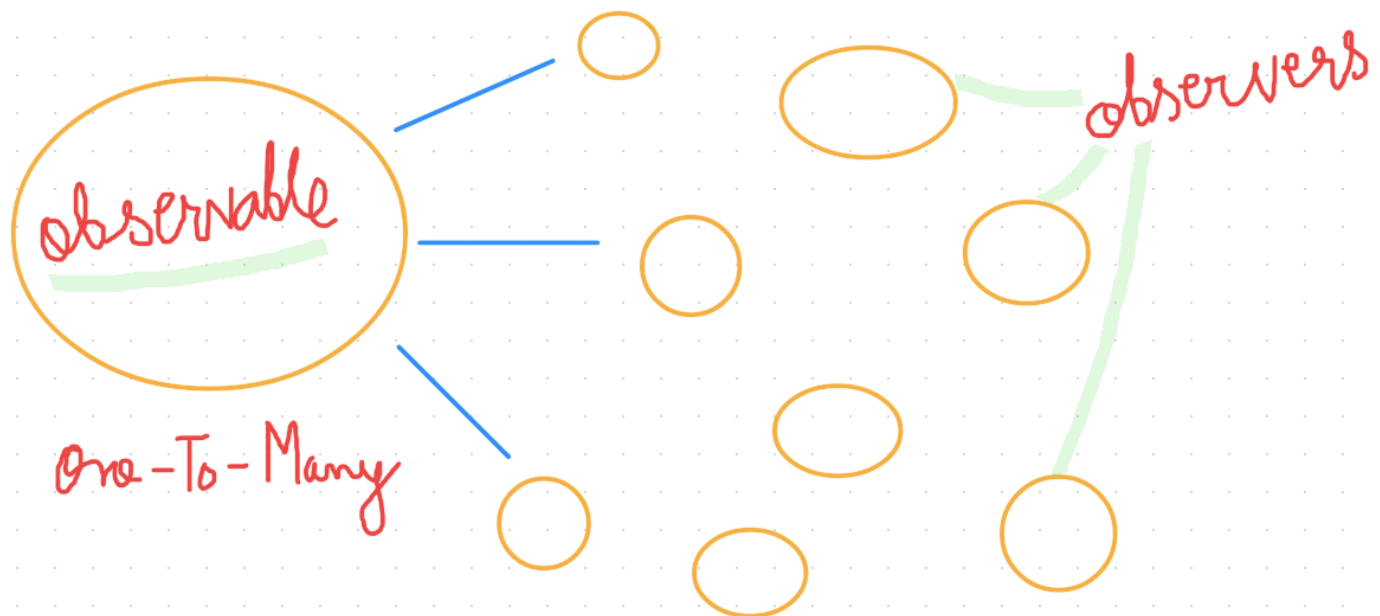


Definition: The **Observer pattern** is a design pattern in which an object, called as an **observable**, maintains a list of its dependents, called **observers**, and **notifies** them automatically of any state changes, usually by calling one of their methods.

Example: A popular use-case of this pattern is the **Notify Me** feature that is present on E-Commerce websites. If a user wants to buy something that is currently out of stock, he can use **Notify Me** feature so that he is **automatically informed** whenever that particular item is back in stock!

Core Concept

There is an **observable** (something that is being observed by others) and there are many **observers** (who observe the observable continuously). There can be **n** number of observers who are interested in knowing some kind of information that the observable holds. Imagine it like this:-



WHAT & WHY?

Taking our previous **E-commerce example**, let's say you want to buy an **Iphone**. You go to an E-commerce website and there you find that Iphone is currently **out of stock**. So, in order to get notified whenever it comes back in stock, you press the **Notify Me** button. Like you, there are **thousands of other users** who are **interested to buy** the Iphone and they too, click the **Notify Me** button.

In this case, you are acting as an **Observer** who is observing the status of availability of Iphone. Similarly, Iphone is acting as an **Observable** (because one or many users are observing it)

When you click on the **Notify Me** button, you get added to the **interested users list** maintained by the Iphone seller. Now, whenever the **Iphone is back in stock**, all the

interested users (**observers**) will be **automatically notified** about it.

PULL v/s PUSH

As, you can notice here, once you have **subscribed** to the **notification service**, you don't have to check the availability of Iphone again and again because you know, you will be automatically notified via email or push notifications whenever Iphone is back in stock.

Now, **let's say**, there was **no notification service**, then you will have to check the availability status of Iphone **again and again** by visiting that E-commerce website. This is called a **PULL** based seeking of information because the user has to **PULL** (make efforts) to know the status.

On, the other hand, **Notification service** is facilitating **PUSH** based mechanism because it will automatically **PUSH** notifications/information to all the users (**observers**) whenever Iphone is back in stock. The **user doesn't need to PULL** (make effort) again and again.

Code Snippets

(Tip: Read through the code comments to get a better perspective)

STEPS:

1.) We create an **Observable Interface**, called **IObservable** which would have the abstract methods as below:

```
IObservable.java × IObservable.java © IphoneObservable.java ©  
1 package observerpattern;  
2  
3 public interface IObservable {  
4     // Add observers who would subscribe to notifications  
5     public void add(IObserver observer);  
6  
7     // Remove observers who would unsubscribe to notifications  
8     public void remove(IObserver observer);  
9  
10    // Notify observers when Iphone is back in stock  
11    public void notifyObservers();  
12  
13    // Set the new stock count for Iphone  
14    public void setStockCount(int stockCount);  
15  
16    // Returns the current stock count for Iphone  
17    public int getStockCount();  
18 }  
19
```

2.) We create an **Observer Interface**, called **IObserver**, which would have the abstract method, as shown below:

① IObserver.java ×

```
1 package observerpattern;
2
3 ①↓ public interface IObserver {
4
5     // Observer will update itself when notified
6 ①↓ public void update();
7 }
8
```

3.)To provide implementation of the IObserver interface, we will create a class, called **IphoneObservable** which will provide concrete definition to the abstract methods of **IObservable** interface.

```
1 package observerpattern;
2
3 import java.util.*;
4
5 public class IphoneObservable implements IObservable{
6     private int stockCount = 0; // Initial Stock Count
7     private List<IObserver> list = new ArrayList<>(); // List to store the observers
8
9     // Add a new observer to the list
10    public void add(IObserver observer){ list.add(observer);}
11
12
13
14    // Remove an observer from the list
15    public void remove(IObserver observer){ list.remove(observer);}
16
17
18
19    /* Set the new stock count and notify the observers
20     if previously Iphone was out of stock */
21    public void setStockCount(int stockCount){
22        if(this.stockCount == 0){
23            this.notifyObservers();
24        }
25
26        this.stockCount=stockCount;
27    }
28
29
30
31
32
33
34    // Returns the current stock count
35    public int getStockCount(){ return this.stockCount;}
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

4.) To provide implementation of the `IObserver` interface, we will create a class, called **User** which will provide concrete definition to the abstract method of **IObserver** interface.

```
© User.java ×
1  package observerpattern;
2
3  public class User implements IObserver{
4      // Reference variable for observable
5      IObservable observable;
6
7      // Assigning the observable object through constructor injection
8      public User(IObservable observable){
9          this.observable = observable;
10     }
11
12     // Perform some operations when the user is notified
13     public void update(){
14         System.out.println("User is notified that iphone is back in stock");
15     }
16 }
17
```

5.) Lastly, to test our implementation, we will **create an Iphone object** which is acting as an **Observable** and we will **create 3 users** who will act as **observers**.

We will then add these 3 users to the notification list of Iphone and then we will set the stock count of Iphone.

```
1 package observerpattern;
2
3 public class Main {
4     public static void main(String[] args) {
5         // Create a new observable
6         IphoneObservable iphone = new IphoneObservable();
7
8         // Creating new Observers
9         User user1 = new User(iphone);
10        User user2= new User(iphone);
11        User user3 = new User(iphone);
12
13        // Adding observers to the notification list
14        iphone.add(user1);
15        iphone.add(user2);
16        iphone.add(user3);
17
18        // Updating stock count for Iphone
19        iphone.setStockCount(10);
20        iphone.setStockCount(0);
21        iphone.setStockCount(20);
22    }
23 }
24
```

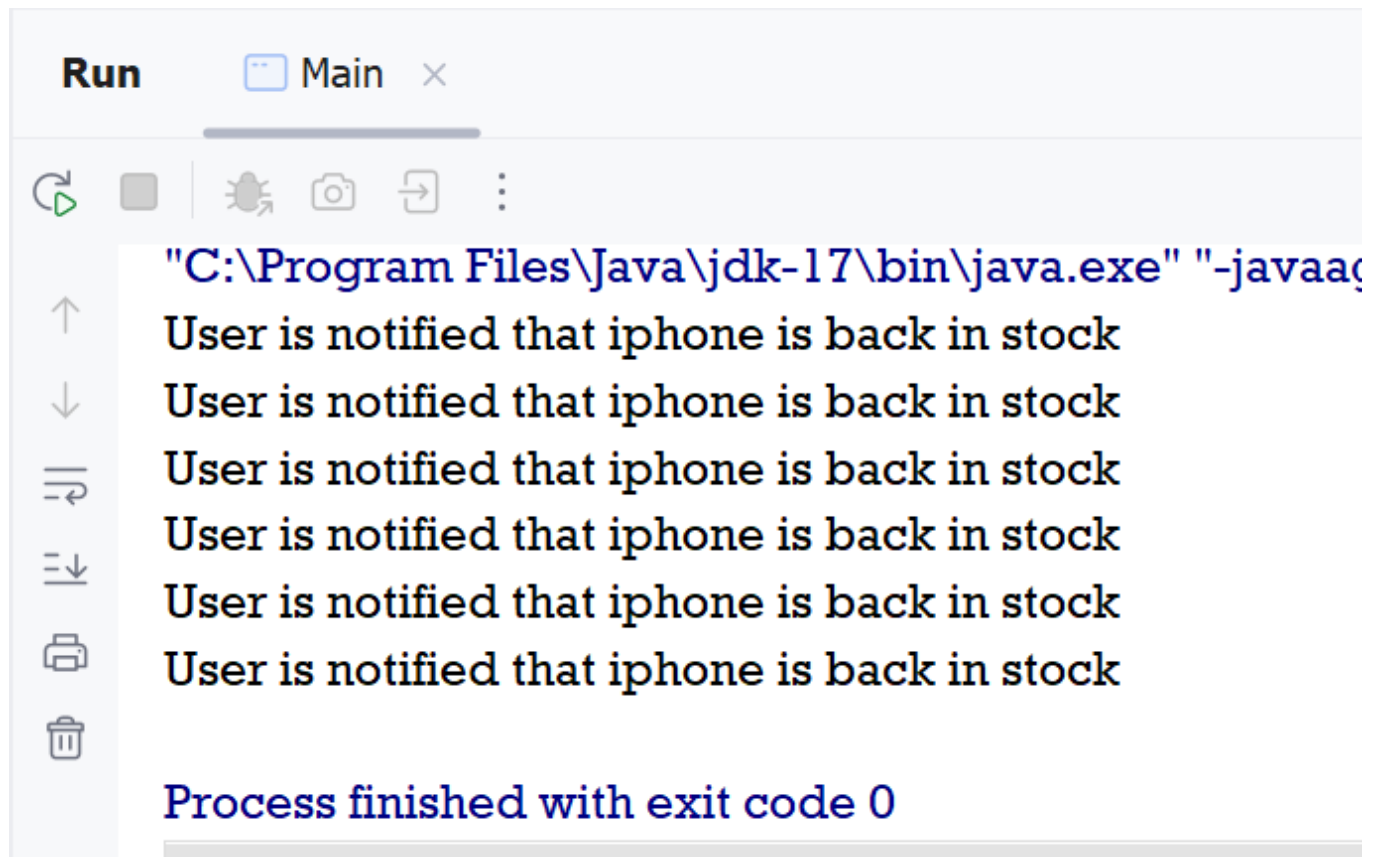
Predicting the output

- a) As the **initial stock count is 0**, so setting the **stock count to 10**, will **notify all the users** about the availability of Iphone.
- b) After that, we will again **set stock count to 0** which **will not notify** the users.

c) Lastly, we will **again set the stock count to 20**, which will **notify the users** again about the availability of Iphone.

So, in total, the **users will be notified twice** about the availability of Iphone when it is back in stock. And, as we have **3 users** in total, **all of them are notified twice**. (That's why, we'll get a **total of 6 lines** of output)

Output:



```
Run Main x
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaa
User is notified that iphone is back in stock
User is notified that iphone is back in stock
User is notified that iphone is back in stock
User is notified that iphone is back in stock
User is notified that iphone is back in stock
User is notified that iphone is back in stock
Process finished with exit code 0
```

Hope you find it helpful. Thanks for reading!



Signing off,
[Bhavuk Jain](#)