

Override clone judiciously

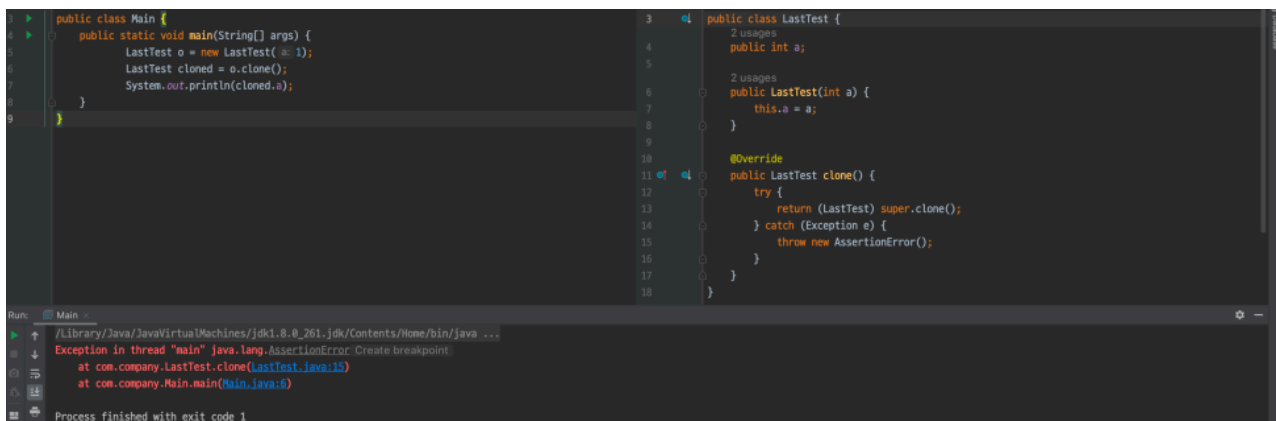
개요

Cloneable interface는 cloning을 허용하는 것을 알리기 위해 만들어졌습니다. 하지만 다음과 같은 문제점들이 있지요.

1. 정작 Cloneable엔 clone 메서드가 없으며 Object에 protected로 clone이 제공되어 문제가 있습니다.
따라서 Cloneable을 구현하는 것만으로는 clone이 작동하리라 보장할 수 없습니다.
2. 리플렉션을 이용해도 clone이 접근가능하게 선언되어있으리라 보장할 수 없어서 문제가 됩니다.

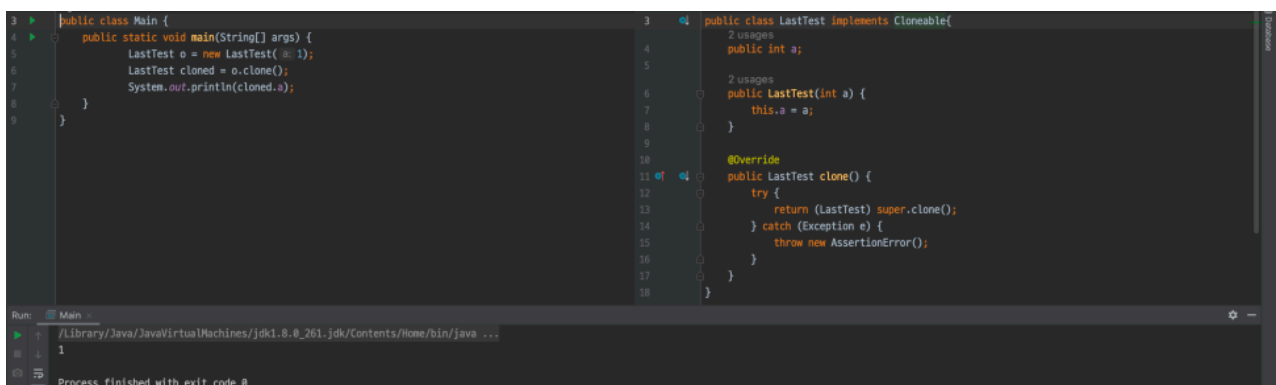
하지만 이러한 문제에도 불구하고 널리 사용되고 있습니다. 따라서 어떻게 clone을 구현해야 하는지, 언제 이렇게 하는 것이 적절한지, 그리고 대안을 말씀드리겠습니다.

cloneable 미구현 실행



```
1 public class Main {
2     public static void main(String[] args) {
3         LastTest o = new LastTest(1);
4         LastTest cloned = o.clone();
5         System.out.println(cloned.a);
6     }
7 }
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
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
```

cloneable구현 실행



```
1 public class Main {
2     public static void main(String[] args) {
3         LastTest o = new LastTest(1);
4         LastTest cloned = o.clone();
5         System.out.println(cloned.a);
6     }
7 }
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
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
```

Cloneable 역할

아무런 메서드없는 cloneable은 clone의 작동방식을 결정합니다. 이를 구현한 클래스의 객체에서 clone을 실행하면 필드를 하나하나씩 복사하여 객체를 반환해줍니다. 그렇지 않으면 ClassNotSupportedException을 던지죠
인터페이스는 클래스의 행동을 정해야 하는데, 자기가 일을 하는 형태는 이례적이므로 이와같이 구현하면 안됩니다.

Cloneable의 허술성

실무관점 문제

Cloneable을 구현하면 clone method를 적절하게 제공할 것처럼 예상되는데, 그렇지 못합니다. 이를 구현하려면 복잡한 규약을 지켜야합니다.

그런데 constructor를 사용하지 않았으므로 위험하고 깨지기 쉬운 프로그램이 되죠.

규약

규약은 강제성이 없습니다. 약한 편이죠.

<https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html> (clone 참조)

그래서 clone method 실행시 super.clone()을 해도 되고, 생성자로 만들어 반환시켜도 됩니다.

하지만! 생성자로 반환하도록 한다면, 이 클래스의 서브클래스에서 clone을 오버라이딩하여 super.clone()을 호출한다면 생성자로 인해 super class의 타입으로 반환되므로 문제가 됩니다.

clone method 상속받아 Cloneable 구현하기

불변 객체 참조시 구현

불변 객체라면 복사하는 것이 무의미하긴 하지만, 이를 구현하면 다음과 같다.

절차

1. super.clone 호출

Object의 clone 메서드를 호출한다.

2. PhoneNumber의 clone 메서드 구현

```
@Override public PhoneNumber clone() {  
    try {  
        return (PhoneNumber) super.clone();  
    } catch (CloneNotSupportedException e) {  
        throw new AssertionError(); // Can't happen  
    }  
}
```

오버라이딩하여 clone을 구현한다. 클라이언트에서 캐스팅하지 않도록 PhoneNumber type으로 반환시켜준다. 자바는 공변타입을 지원하기에 가능하다.

super.clone()이 try-catch로 감싼 이유는 clone실행시 CloneNotSupportedException이란 checked exception이 발생하기 때문이다.

그런데 cloneable을 구현해서 이미 잘 작동되리라 알고 있기 때문에 지저분해진 코드의 예시라 볼 수 있다.

3. Cloneable 구현

clone메서드가 정상작동되기 위해 Cloneable을 구현하도록 한다.

가변객체 참조시 구현 (여러가지 해결 방법을 보여줌)

불변 객체를 clone할 시엔 문제가 없는데, 가변객체를 참조한다면 문제가 된다.

문제되는 예시

```
public class Stack {
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;
    public Stack() {
        this.elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }
    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }
    public Object pop() {
        if (size == 0)
            throw new EmptyStackException();
        Object result = elements[--size];
        elements[size] = null; // Eliminate obsolete reference
        return result;
    }
    // Ensure space for at least one more element.
    private void ensureCapacity() {
        if (elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }
}
```

이놈을 보면 elements를 복사할 땐 안의 값들은 shallow copy가 된다. 따라서 Stack 객체를 생성하더라도 elements의 동일한 객체의 주소를 바라보기 때문에 문제가 된다.

만약에 clone method가 단일 constructor을 제공했다면 문제가 되지않았을 것이다. 다시 말해서 clone method는 origin object에 영향이 없고, 불변 객체를 적절하게 생성해야 한다.

이를 달성하기 위해선 stack의 내부 값을 복사하도록 하면 된다.

해결 방법

```
// Clone method for class with references to mutable state
@Override public Stack clone() {
    try {
        Stack result = (Stack) super.clone();
        result.elements = elements.clone();
        return result;
    } catch (CloneNotSupportedException e) {
        throw new AssertionError();
    }
}
```

주안점

1. elements의 compile time과 run time의 타입이 같기 때문에 캐스팅할 필요 없습니다.
이 때문에 배열을 복사할 때는 clone메서드를 유일하게 사용하기 적합한 예라고 볼 수 있습니다.
2. elements는 가변객체이기에 일반 관례를 따른다면 final로 구현될 수도 있지만, 위처럼 복사하고 싶은 경우엔 제거하여 구현하면 된다.

또 다른 문제

재귀 호출로만 충분하지 않은 예도 있습니다.

해쉬 테이블용 clone method를 예시로 들어보겠습니다.

```
public class HashTable implements Cloneable {
    private Entry[] buckets = ...;

    private static class Entry {
        final Object key;
        Object value;
        Entry next;

        Entry(Object key, Object value, Entry next) {
            this.key = key;
            this.value = value;
            this.next = next;
        }
    }
}
```

```
// Broken clone method - results in shared mutable state!
@Override
public HashTable clone() {
    try {
        HashTable result = (HashTable) super.clone();
        result.buckets = buckets.clone();
        return result;
    } catch (CloneNotSupportedException e) {
        throw new AssertionError();
    }
}
}
```

해쉬 테이블에 element로 linked-list가 들어가있는 형태를 봅시다. 이것이 버킷이 됩니다.

위처럼 clone을 한다면 내부가 또 리스트이기 때문에 결국 shallow copy를 할 수 밖에 없죠.

mutable object 구현 - recursion

따라서 내부에 deepcopy를 구현해야 하죠.

```
public class HashTable implements Cloneable {
    private Entry[] buckets = ...;

    private static class Entry {
        final Object key;
        Object value;
        Entry next;

        Entry(Object key, Object value, Entry next) {
            this.key = key;
            this.value = value;
            this.next = next;
        }

        // Recursively copy the linked list headed by this Entry
        Entry deepCopy() {
            return new Entry(key, value,
                next == null ? null : next.deepCopy());
        }
    }

    @Override
    public HashTable clone() {
        try {
            HashTable result = (HashTable) super.clone();
```

```

        result.buckets = new Entry[buckets.length];
        for (int i = 0; i < buckets.length; i++)
            if (buckets[i] != null)
                result.buckets[i] = buckets[i].deepCopy();
        return result;
    } catch (CloneNotSupportedException e) {
        throw new AssertionError();
    }
}
... // Remainder omitted
}

```

하지만 이와 같이 구현한다면 버킷이 길다면 stack overflow가 발생할 여지가 있습니다. 따라서 iteration을 통해 개선할 수 있죠.

mutable object 구현 - iteration

```

Entry deepCopy() {
    Entry result = new Entry(key, value, next);
    for (Entry p = result; p.next != null; p = p.next)
        p.next = new Entry(p.next.key, p.next.value, p.next.next);
    return result;
}

```

mutable object 구현 - super.clone

고수준 API를 이용하여 clone을 구현하는 방법

1. super.clone으로 필드 초기화
2. put(key, value)같은 고수준 method를 통해 deep copy를 진행해줍니다.

매우 간단하게 만들어 주지만 속도가 느리고, cloneable이 field를 단계단계 생성해준다는 architecture에는 어울리지 않게 됩니다.

주의 사항

1. clone은 overridable method를 호출하면 안됩니다.

하위 클래스에서 clone에서 호출했던 method를 오버라이딩한다면 고칠 수 있겠죠. 따라서 supe.clone에서 put(key, value) method를 호출했다면 이것은 private이나 final로 선언하여 오버라이딩을 막아야 합니다.

2. overriding method는 CloneNotSupportedException을 던지지 않도록 합니다.

Object에서는 clone not supportedException을 던지나, sub class에서는 public으로 선언하고 checked exception을 던지지 않도록 합니다. 사용의 편리성을 위해서지요.

3. clone 상속 구현 방법

공통적으로 Cloneable을 구현하면 안됩니다.

1) Object방식 모방

clone을 제대로 구현한 뒤 protected로 선언. 그리고 CloneNotSupprtedException을 던지도록 한다.

Object처럼 cloneable 구현여부를 sub class에서 선택할 수 있도록 함.

2) clone 막기

final로 막고 에러를 던지도록 합니다.

```
@Override
protected final Object clone() throws CloneNotSupportedException {
    throw new CloneNotSupportedException();
}
```

4. thread-safe class로 만들 것.

clone은 synchronized를 고려하지 않았기 때문에 이를 고려해주어야 한다.

이 복잡한 과정들이 모두 필요할까?

이러한 경우는 드뭅니다.

단지 cloneable을 확장한 클래스를 상속받는다면 어쩔수없이 clone을 구현해주어야 합니다.

이 외 대다수 경우의 접근은 copy constructor나 copy factory를 제공해주는 것입니다. (더 정확한 이름은 conversion constructor, conversion factory)

copy constructor는 단순히 자기 자신의 필드와 동일한 필드를 생성해주는 생성자를 의미합니다.

```
// Copy constructor
public Yum(Yum yum) { ... };
A copy factory is the static factory (Item 1) analogue of a copy constructor:
// Copy factory
public static Yum newInstance(Yum yum) { ... };
```

이점

1. 언어 모순적이고 위험천만한 객체 생성 매커니즘 사용 안함
2. 영성한 문서 규약 기대지 않음
3. 정상적인 final field용법과 충돌되지 않음
4. 불필요한 checked exception을 던지지 않음
5. casting도 필요없음
6. interface 타입의 인스턴스를 인수로 받을 수 있다.

이로 인해 HashSet 객체를 TreeSet으로 변환 가능