



Exercise 44: Inheritance Versus Composition

In the fairy tales about heroes defeating evil villains there's always a dark forest of some kind. It could be a cave, a forest, another planet, just some place that everyone knows the hero shouldn't go. Of course, shortly after the villain is introduced you find out, yes, the hero has to go to that stupid forest to kill the bad guy. It seems the hero just keeps getting into situations that require him to risk his life in this evil forest.

You rarely read fairy tales about the heroes who are smart enough to just avoid the whole situation entirely. You never hear a hero say, "Wait a minute, if I leave to make my fortunes on the high seas leaving Buttercup behind I could die and then she'd have to marry some ugly prince named Humperdink. Humperdink! I think I'll stay here and start a Farm Boy for Rent business." If he did that there'd be no fire swamp, dying, reanimation, sword fights, giants, or any kind of story really. Because of this, the forest in these stories seems to exist like a black hole that drags the hero in no matter what they do.

In object-oriented programming, Inheritance is the evil forest. Experienced programmers know to avoid this evil because they know that deep inside the Dark Forest Inheritance is the Evil Queen Multiple Inheritance. She likes to eat software and programmers with her massive complexity teeth, chewing on the flesh of the fallen. But the forest is so powerful and so tempting that nearly every programmer has to go into it, and try to make it out alive with the Evil Queen's head before they can call themselves real programmers. You just can't resist the Inheritance Forest's pull, so you go in. After the adventure you learn to just stay out of that stupid forest and bring an army if you are ever forced to go in again.

This is basically a funny way to say that I'm going to teach you something you should use carefully called Inheritance. Programmers who are currently in the forest battling the Queen will probably tell you that you have to go in. They say this because they need your help since what they've created is probably too much for them to handle. But you should always remember this:

Most of the uses of inheritance can be simplified or replaced with composition, and I'll show you how in this exercise.

What is Inheritance?

Inheritance is used to indicate that one class will get most or all of its features from a parent class. This happens implicitly whenever you write `class Foo < Bar`, which says "Make a class Foo that inherits from Bar." When you do this, the language makes any action that you do on instances of `Foo` also work as if they were done to an instance of `Bar`. Doing this lets you put common functionality in the `Bar` class, then specialize that functionality in the `Foo` class as needed.

When you are doing this kind of specialization, there are three ways that the



Follow

parent and child classes can interact:

1. Actions on the child imply an action on the parent.
2. Actions on the child override the action on the parent.
3. Actions on the child alter the action on the parent.

I will now demonstrate each of these in order and show you code for them.

Implicit Inheritance

First I will show you the implicit actions that happen when you define a function in the parent, but *not* in the child.

```
1 class Parent
2
3   def implicit()
4     puts "PARENT implicit()"
5   end
6 end
7
8 class Child < Parent
9 end
10
11 dad = Parent.new()
12 son = Child.new()
13
14 dad.implicit()
15 son.implicit()
```

1
2
3
4
5
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

MAIN



PLAY VIDEO



PREVIOUS



NEXT



HELP

Follow

The class `Child` is an empty class that inherits all of its behavior from `Parent`. When you this code you get the following:

```
$ ruby ex44a.rb
PARENT implicit()
PARENT implicit()
```

Notice how even though I'm calling `son.implicit()` on line 16, and even though `Child` does *not* have a `implicit` function defined, it still works and it calls the one defined in `Parent`. This shows you that, if you put functions in a base class (i.e., `Parent`) then all subclasses (i.e., `Child`) will automatically get those features. Very handy for repetitive code you need in many classes.

Override Explicitly

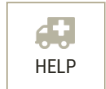
The problem with having functions called implicitly is sometimes you want the child to behave differently. In this case you want to override the function in the child, effectively replacing the functionality. To do this just define a function with the same name in `Child`. Here's an example:

```
1 class Parent
2
3   def override()
4     puts "PARENT override()"
5   end
6 end
7
8 class Child < Parent
```

```

9   def override()
10     puts "CHILD override()"
11   end
12 end
13
14 dad = Parent.new()
15 son = Child.new()
16
17 dad.override()
18 son.override()

```



Follow

In this example I have a function named `override` in both classes, so let's see what happens when you run it.

```

$ ruby ex44b.rb
PARENT override()
CHILD override()

```

As you can see, when line 14 runs, it runs the `Parent.override` function because that variable (`dad`) is a `Parent`. But when line 15 runs it prints out the `Child.override` messages because `son` is an instance of `Child` and `Child` overrides that function by defining its own version.

Take a break right now and try playing with these two concepts before continuing.

Alter Before or After

The third way to use inheritance is a special case of overriding where you want to alter the behavior before or after the `Parent` class's version runs. You first override the function just like in the last example, but then you use a Ruby built-in function named `super` to get the `Parent` version to call. Here's the example of doing that so you can make sense of this description:

```

1   class Parent
2     def altered()
3       puts "PARENT altered()"
4     end
5   end
6
7   class Child < Parent
8     def altered()
9       puts "CHILD, BEFORE PARENT altered()"
10      super()
11      puts "CHILD, AFTER PARENT altered()"
12    end
13
14  end
15
16  dad = Parent.new()
17  son = Child.new()
18
19  dad.altered()
20  son.altered()

```

The important lines here are 9-11, where in the `Child` I do the following when `son.altered()` is called:

1. Because I've overridden `Parent.altered` the `Child.altered`

- version runs, and line 9 executes like you'd expect.
2. In this case I want to do a before and after so after line 9, I want to use `super` to get the `Parent.altered` version.
 3. On line 10 I call `super()`, which is aware of inheritance and will get the `Parent` class for you.
 4. At this point, the `Parent.altered` version of the function runs, and that prints out the `Parent` message.
 5. Finally, this returns from the `Parent.altered` and the `Child.altered` function continues to print out the after message.

If you run this, you should see this:

```
$ ruby ex44c.rb
PARENT altered()
CHILD, BEFORE PARENT altered()
PARENT altered()
CHILD, AFTER PARENT altered()
```

All Three Combined

To demonstrate all of these, I have a final version that shows each kind of interaction from inheritance in one file:

```
1 class Parent
2
3   def override()
4     puts "PARENT override()"
5   end
6
7   def implicit()
8     puts "PARENT implicit()"
9   end
10
11  def altered()
12    puts "PARENT altered()"
13  end
14 end
15
16 class Child < Parent
17
18   def override()
19     puts "CHILD override()"
20   end
21
22   def altered()
23     puts "CHILD, BEFORE PARENT altered()"
24     super()
25     puts "CHILD, AFTER PARENT altered()"
26   end
27 end
28
29 dad = Parent.new()
30 son = Child.new()
31
32 dad.implicit()
33 son.implicit()
34
35 dad.override()
36 son.override()
37
38 dad.altered()
```

1
2
3
4
5
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



Follow

Go through each line of this code, and write a comment explaining what that line does and whether it's an override or not. Then run it and confirm you get what you expected:

```
$ ruby ex44d.rb
PARENT implicit()
PARENT implicit()
PARENT override()
CHILD override()
PARENT altered()
CHILD, BEFORE PARENT altered()
PARENT altered()
CHILD, AFTER PARENT altered()
```

Using `super()` with `initialize`

The most common use of `super()` is actually in `initialize` functions in base classes. This is usually the only place where you need to do some things in a child, then complete the initialization in the parent. Here's a quick example of doing that in the `Child` from these examples:

```
class Child < Parent
  def initialize(stuff)
    @stuff = stuff
    super()
  end
end
```

This is the same as the `Child.altered` example above, except I'm setting some variables in the `initialize` before having the `Parent` initialize with its `Parent.initialize`.

Composition

Inheritance is useful, but another way to do the exact same thing is just to *use* other classes and modules, rather than rely on implicit inheritance. If you look at the three ways to exploit inheritance, two of the three involve writing new code to replace or alter functionality. This can easily be replicated by just calling functions in a module. Here's an example of doing this:

```
1 class Other
2
3   def override()
4     puts "OTHER override()"
5   end
6
7   def implicit()
8     puts "OTHER implicit()"
9   end
10
11  def altered()
12    puts "OTHER altered()"
13  end
14 end
15
16 class Child
17
```


 MAIN


 PLAY VIDEO


 PREVIOUS


 NEXT


 HELP

Follow

```

18   def initialize()
19     @other = Other.new()
20   end
21
22   def implicit()
23     @other.implicit()
24   end
25
26   def override()
27     puts "CHILD override()"
28   end
29
30   def altered()
31     puts "CHILD, BEFORE OTHER altered()"
32     @other.altered()
33     puts "CHILD, AFTER OTHER altered()"
34   end
35 end
36
37 son = Child.new()
38
39 son.implicit()
40 son.override()
41 son.altered()

```



Follow

In this code I'm not using the name `Parent`, since there is *not* a parent-child `is-a` relationship. This is a `has-a` relationship, where `Child` `has-a` `Other` that it uses to get its work done. When I run this I get the following output:

```

$ ruby ex44e.rb
OTHER implicit()
CHILD override()
CHILD, BEFORE OTHER altered()
OTHER altered()
CHILD, AFTER OTHER altered()

```

You can see that most of the code in `Child` and `Other` is the same to accomplish the same thing. The only difference is that I had to define a `Child.implicit` function to do that one action. I could then ask myself if I need this `Other` to be a class, and could I just make it into a module named `other.rb`?

Ruby has another way to do composition using modules and a concept called mixins. You simply create a module with functions that are common to classes and then include them in your class similar to using a `require`. Here's this same composition example done using modules and mixins.

```

1  module Other
2
3    def override()
4      puts "OTHER override()"
5    end
6
7    def implicit()
8      puts "OTHER implicit()"
9    end
10
11   def Other.altered()
12     puts "OTHER altered()"
13   end
14 end
15
16 class Child

```

```

17  include Other
18
19  def override()
20    puts "CHILD override()"
21  end
22
23  def altered()
24    puts "CHILD, BEFORE OTHER altered()"
25    Other.altered()
26    puts "CHILD, AFTER OTHER altered()"
27  end
28 end
29
30 son = Child.new()
31
32 son.implicit()
33 son.override()
34 son.altered()

```

This is similar to the previous composition example. Mixins are much more powerful and an advanced topic I won't cover in this book.

When to Use Inheritance or Composition

The question of "inheritance versus composition" comes down to an attempt to solve the problem of reusable code. You don't want to have duplicated code all over your software, since that's not clean and efficient. Inheritance solves this problem by creating a mechanism for you to have implied features in base classes. Composition solves this by giving you modules and the ability to call functions in other classes.

If both solutions solve the problem of reuse, then which one is appropriate in which situations? The answer is incredibly subjective, but I'll give you my three guidelines for when to do which:

1. Avoid something called "meta-programming" at all costs, as it is too complex to be useful reliably. If you're stuck with it, then be prepared to know the class hierarchy and spend time determining where everything is coming from.
2. Use composition to package up code into modules that are used in many different unrelated places and situations.
3. Use inheritance only when there are clearly related reusable pieces of code that fit under a single common concept or if you have to because of something you're using.

Do not be a slave to these rules. The thing to remember about object-oriented programming is that it is entirely a social convention programmers have created to package and share code. Because it's a social convention, but one that's codified in Ruby, you may be forced to avoid these rules because of the people you work with. In that case, find out how they use things and then just adapt to the situation.

Study Drills



Follow

There is only one Study Drill for this exercise because it is a big exercise. Go and read <https://github.com/bbatsov/ruby-style-guide> and start trying to use it in your code. You'll notice that some of it is different from what you've been learning in this book, but now you should be able to understand their recommendations and use them in your own code. The rest of the code in this book may or may not follow these guidelines depending on if it makes the code more confusing. I suggest you also do this, as comprehension is more important than impressing everyone with you knowledge of esoteric style rules.



Follow

Common Student Questions

Q: How do I get better at solving problems that I haven't seen before?

The only way to get better at solving problems is to solve as many problems as you can *by yourself*. Typically people hit a difficult problem and then rush out to find an answer. This is fine when you have to get things done, but if you have the time to solve it yourself, then take that time. Stop and bang your head against the problem for as long as possible, trying every possible thing, until you solve it or give up. After that the answers you find will be more satisfying and you'll eventually get better at solving problems.

Q: Aren't objects just copies of classes?

In some languages (like JavaScript) that is true. These are called prototype languages and there are not many differences between objects and classes other than usage. In Ruby, however, classes act as templates that "mint" new objects, similar to how coins were minted using a die (template).

Video

Learn Ruby The Hard Way, 3rd Edition

When you buy the book from me for **29.59** you get all of the following:

- PDF of the book updated when the site updates.
- Access to the site with no ads.
- 6G of video you can download and watch, one for each exercise.
- Video in 720p high quality HD format.
- No DRM on any content.
- Updates until the next edition is released.

Buying Is Easy

Buying is easy. Just fill out the form below and we'll get started.

Full Name

Email Address

☐ Pay With Credit Card (by Stripe™)

☐ Use your PayPal™ account.

Buy Learn Ruby The Hard Way, 3rd Edition

Already Paid? Reactivate Your Purchase Right Now!

Zed Shaw

PDF + Videos +
Updates

Amazon

Paper + DVD
\$26.67

Zed Shaw

PDF + Videos +
Updates

Amazon

Paper + DVD
\$26.67

Interested In Python?

Python is also a great language.
Learn Python The Hard Way



\$29.95



\$29.95



1

2

3

4

MAIN

PLAY VIDEO

PREVIOUS

NEXT

HELP

Follow