

UT 6 y 7: Desarrollo de clases. Clases avanzadas

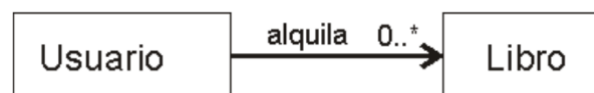
1.	<i>Relaciones entre clases</i>	2
1.1.	Implementación de composición y agregación en Java	2
2.	<i>Herencia</i>	3
2.1.	Casting entre clases	7
2.2.	InstanceOf	9
3.	<i>Clase abstracta</i>	10
4.	<i>Clases finales</i>	11
5.	<i>Interfaces</i>	12
5.1.	Interfaz vs clase abstracta	13
6.	<i>Clases internas (Inner Class)</i>	13

1. Relaciones entre clases

Aunque en temas anteriores se han visto las relaciones entre clases, en este tema, se van a desarrollar desde un punto de vista más teórico para facilitar el diseño de las aplicaciones.

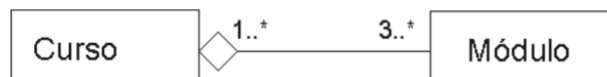
Se pueden distinguir varios tipos de relaciones:

- **Asociaciones:** se tratan de relaciones directas entre clases. Dos clases tienen una asociación si:
 - Un objeto envía un mensaje a otro objeto. Enviar un mensaje, como ya se comentó es utilizar alguno de sus métodos o propiedades para que el objeto realice una determinada labor.
 - Un objeto instancia a otra clase (crea un objeto de otra clase).
 - Un objeto recibe como parámetros de un método objetos de otra clase.



```
public class Usuario {  
    private Libro libro; // Cardinalidad 0..1  
    private Libro libros[]; // Cardinalidad 0..*  
    private List<Libro> librosLista; // Cardinalidad 0..*
```

- **Agregación:** Indica que un elemento es parte de otro. Así la clase Curso tendría una relación de composición con la clase Módulo.



Cada curso se compone de tres o más módulos. Cada módulo se relaciona con uno o más cursos.

- **Composición:** se trata de una relación más fuerte que el caso anterior. El principio de esta relación es que el tiempo de vida de la clase contenedora y la clase contenida coinciden. Básicamente, si un objeto contenedor se destruye, el objeto contenido también.



1.1. Implementación de composición y agregación en Java

La implementación en Java de clases con relaciones de agregación y composición es similar. Pero hay un matiz importante. Puesto que en la composición los

objetos que se usan para componer el objeto contenedor tienen una existencia ligada al mismo, se deben crear dentro del objeto contenedor. Por ejemplo (composición):

```
public class Edificio {  
    private Piso piso[];  
    public Edificio(int numPisos){  
        piso=new Piso[numPisos]; //composición  
    }  
}
```

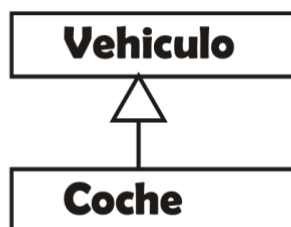
la existencia del piso está ligada al edificio por eso los pisos del edificio se deben de crear dentro de la clase Edificio y así cuando un objeto Edificio desaparezca, desaparecerán los pisos del mismo.

Eso no debe ocurrir si la relación es de agregación. Por eso en el caso de los cursos y los módulos, como los módulos no tienen esa dependencia de existencia según el diagrama, seguirán existiendo cuando el módulo desaparezca, por eso se deben declarar fuera de la clase cursos. Es decir, no habrá new para crear módulos en el constructor.

```
public class Curso {  
    private Modulo modulos[];  
    public Curso(Modulo modulos[]) {  
        this.modulos = modulos; // agregación  
    }  
}
```

2. Herencia

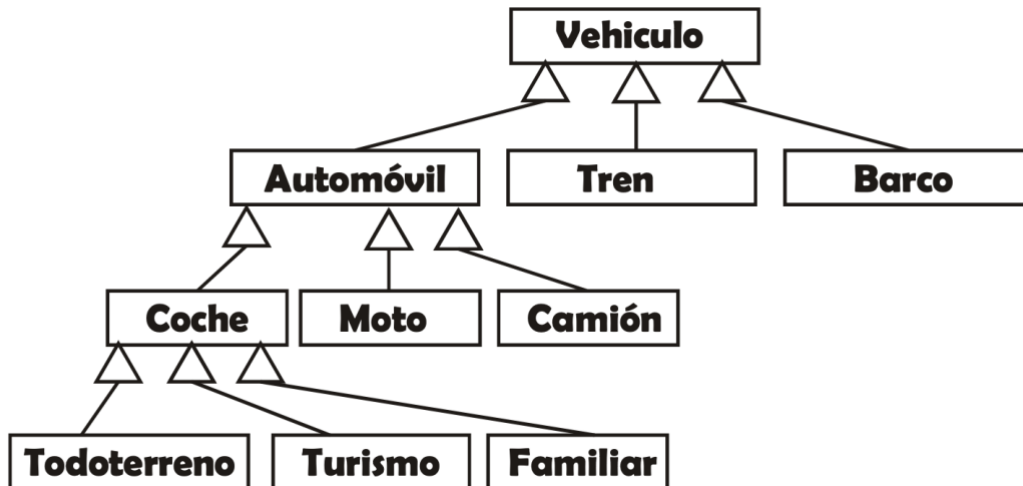
Para que una clase herede las características de otra hay que utilizar la palabra clave *extends* tras el nombre de la clase. A esta palabra le sigue el nombre de la clase cuyas características se heredarán. En Java solo se puede tener herencia de una clase.



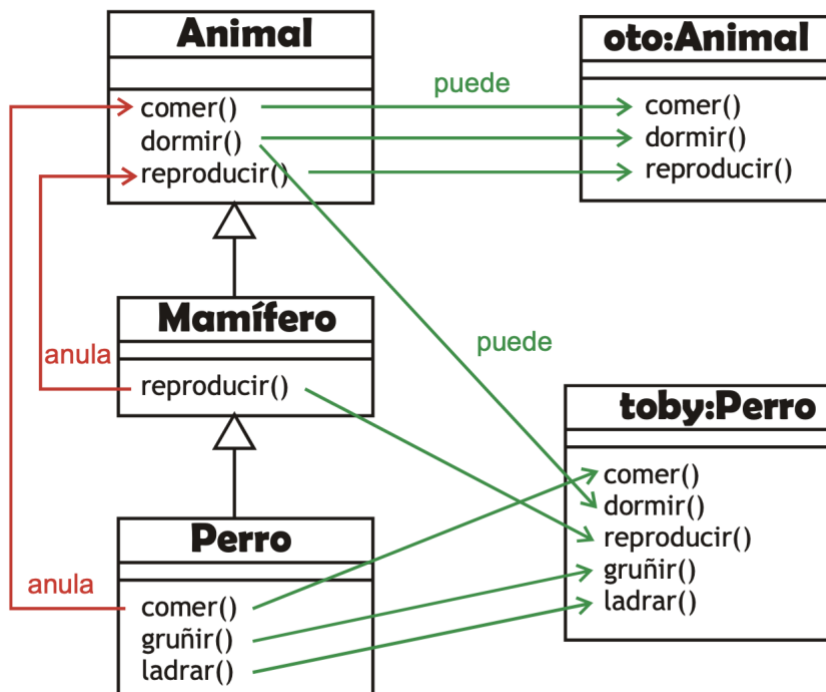
```
public class Coche extends Vehiculo {  
  
}
```

El diagrama representa que un Coche es un Vehículo. Con lo cual los coches tendrán todas las características de los vehículos y además añadirán características particulares.

Además, puede haber varios niveles de herencia.



Es muy importante tener en cuenta que la herencia suele ir ligada a otro concepto importante de POO, el polimorfismo. Gracias a este último, una clase hija puede heredar un método y sobre-escribirlo, o dicho de otra forma, puede “hacerlo suyo”.



En este ejemplo, *oto* y *toby* son instancias de *Animal* y *Perro*, respectivamente. En el caso de *toby*, al ser una instancia de *Perro*, utiliza los métodos “comer”, “gruñir” y

“ladrar” de esa clase, “dormir” de Animal y “reproducir” de Mamífero (herencia). En el caso de *oto*, todos sus métodos son los de la clase Animal.

También se pueden utilizar los métodos de la clases padre, mediante el uso de *super*:

```
public class Vehiculo {
    public double velocidad;
    public void acelerar(double kmh) {
        velocidad += kmh;
    }
}

public class Coche extends Vehiculo {
    public int gasolina;
    public void acelerar(double kmh) {
        super.acelerar(kmh);
        gasolina *= 0.9;
    }
}
```

En este caso, cuando se utiliza el método “acelerar” de Coche, primero se invoca al mismo método de la superclase, para rellenar el atributo velocidad y luego se modifica el atributo gasolina. El atributo velocidad lo hereda también Coche, puesto que *public*.

IMPORTANTE: los constructores no se heredan de la clase base a las clases derivadas, pero sí se puede invocar al constructor de la clase base.

```
public class A {
    protected int valor;
    public A(int v) {
        valor = v;
    }
    public void escribir() {
        System.out.println(valor);
    }
}

public class B extends A {
    public B(int v) {
        super(v);
    }
    public void escribir() {
        System.out.println(valor * 2);
    }
}
```

En el caso de B, siempre se debe invocar al constructor del padre en la primera línea del constructor de esa misma clase (en todos los constructores que disponga). En caso contrario, daría error:

```

 4
 3 public class B extends A {
 4
 5 public B(int v) {
 6 //     super(v);
 7     valor = v;
 8 }
 9
10 public void escribir() {
11     System.out.println(valor * 2);
12 }
13

```

Un ejemplo de ejecución sería:

```

 4
 3 public class PruebaAB {
 4
 5
 6 public static void main(String[] args) {
 7     A a = new A(2);
 8     a.escribir();
 9
10     B b = new B(2);
11     b.escribir();
12 }
13
14 }

```

Problems @ Javadoc Declaration Search Console

<terminated> PruebaAB [Java Application] /Users/fran/.p2/pool/plugins/or

2

4

En caso de no necesitar constructores, Java los crea por defecto. En el caso de la instancia de la clase hija, Java invocaría al constructor de la clase padre, de forma implícita.

```

 4
 3 public class A {
 4     protected int valor;
 5
 6 public void setValor(int valor) {
 7     this.valor = valor;
 8 }
 9
10 public void escribir() {
11     System.out.println(valor);
12 }
13
14 }
15

```

```

 4
 3 public class B extends A {
 4
 5 public void escribir() {
 6     System.out.println(valor * 2);
 7 }
 8
 9 }
10

```

```

 4
 3 public class PruebaAB {
 4
 5
 6 public static void main(String[] args) {
 7     A a = new A();
 8     a.setValor(2);
 9     a.escribir();
10
11     B b = new B();
12     b.setValor(2);
13     b.escribir();
14 }
15
16 }
17

```

Problems @ Javadoc Declaration Search Console

<terminated> PruebaAB [Java Application] /Users/fran/.p2/pool/plugins/or

2

4

Ojo, si la clase padre dispone de un constructor, la hija también deberá implementarlo.

```

3 public class A {
4     protected int valor;
5
6     public A(int valor) {
7         this.valor = valor;
8     }
9
10    public void escribir() {
11        System.out.println(valor);
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

```

En este caso, B daría error porque no tiene un constructor con la llamada al constructor del padre (*super*).

2.1. Casting entre clases

El proceso es similar al de los tipos primitivos en cuanto a la implementación. Hay varios casos:

- Una clase padre se puede “convertir” en una clase hija.
- Una clase hija se puede “convertir” en la clase padre mediante un casting.

```

3 public class Vehiculo {
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

```

En el siguiente caso, convertimos un coche en un vehículo.

```

1
2
3 public class PruebaVehiculo {
4
5     public static void main(String[] args) {
6         Vehiculo v1 = new Vehiculo();
7         Coche c1 = new Coche("Audi");
8         c1.repostar(50);
9         System.out.println(c1); // Se invoca de forma automática el método toString
10        v1 = c1;
11        System.out.println(v1); // Se invoca de forma automática el método toString
12    }
13 }
14
15 }

```

Problems @ Javadoc Declaration Search Console Servers Error Log Git Status

<terminated> PruebaVehiculo (1) [Java Application] /Users/fran/.p2/pool/plugins/org.eclipse.justj.openjdk.hotspot.jre

Coche
 Marca: Audi
 Depósito: 50litros
 Coche
 Marca: Audi
 Depósito: 50litros

Si quisiera convertir una clase padre en una hija, sin el casting daría error:

```

1
2
3 public class PruebaVehiculo {
4
5     public static void main(String[] args) {
6         Vehiculo v1 = new Vehiculo();
7         Coche c1 = new Coche("Audi");
8         c1.repostar(50);
9         c1 = v1;
10    }
11 }
12
13 }

```

Sin embargo, si uso un casting, no habría problema en cuanto a la implementación, pero daría un error:

```

1
2
3 public class PruebaVehiculo {
4
5     public static void main(String[] args) {
6         Vehiculo v1 = new Vehiculo();
7         Coche c1 = new Coche("Audi");
8         c1 = (Coche)v1;
9         c1.repostar(50);
10        System.out.println(c1);
11    }
12 }

```

Problems @ Javadoc Declaration Search Console Servers Error Log Git Status

<terminated> PruebaVehiculo (1) [Java Application] /Users/fran/.p2/pool/plugins/org.eclipse.justj.openjdk.hotspot.jre

Exception in thread "main" java.lang.ClassCastException: class com.venancio.dam.tema6.PruebaVehiculo cannot be cast to class com.venancio.dam.tema6.Coche (com.venancio.dam.tema6.PruebaVehiculo is not a subclass of com.venancio.dam.tema6.Coche)

La única forma de que no dé error sería que v1 hiciera referencia a un Coche:


```

4
3 public class PruebaVehiculo {
4
5     public static void main(String[] args) {
6         Vehiculo v1 = new Coche("Audi");
7         Coche c1 = (Coche)v1;
8         c1.repostar(50);
9         System.out.println(c1);
10    }
11 }
12

```

Problems @ Javadoc Declaration Search Console

<terminated> PruebaVehiculo (1) [Java Application] /Users/fran/.p2/pool/p

Coche
Marca: Audi
Depósito: 50litros

De hecho, ni siquiera necesitamos almacenar la referencia de coche, simplemente con el uso del casting podríamos acceder a los miembros de la clase hija:

```

4
3 public class PruebaVehiculo {
4
5     public static void main(String[] args) {
6         Vehiculo v1 = new Coche("Audi");
7         ((Coche)v1).repostar(50);
8         System.out.println(v1);
9     }
10 }
11

```

Problems @ Javadoc Declaration Search Console

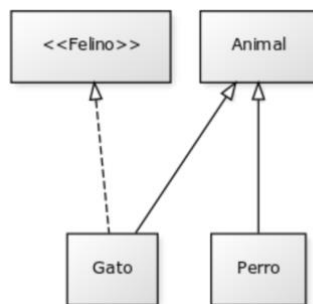
<terminated> PruebaVehiculo (1) [Java Application] /Users/fran/.p2/pool/p

Coche
Marca: Audi
Depósito: 50litros

2.2. InstanceOf

Permite comprobar si un determinado objeto pertenece a una clase concreta.

Ejemplo:



```

public interface Felino {
    |
}

```

```

public class Animal {
}

```

```

public class Gato extends Animal implements Felino{
}

```

```

public class Perro extends Animal{
}

```

```

public class OperadorInstanceOf {
    public static void main(String[] args) {
        Animal simba = new Animal();
        Perro idefix = new Perro();
        Gato garfield = new Gato();

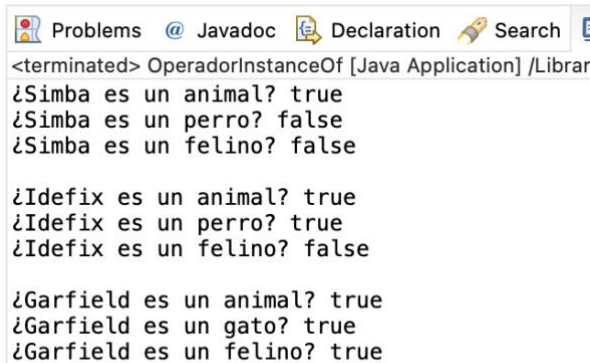
        System.out.println("¿Simba es un animal? " + (simba instanceof Animal)); //true
        System.out.println("¿Simba es un perro? " + (simba instanceof Perro)); //false
        System.out.println("¿Simba es un felino? " + (simba instanceof Felino)); //false

        System.out.println("\n¿Idex es un animal? " + (idefix instanceof Animal)); //true
        System.out.println("¿Idex es un perro? " + (idefix instanceof Perro)); //true
        System.out.println("¿Idex es un felino? " + (idefix instanceof Felino)); //false

        System.out.println("\n¿Garfield es un animal? " + (garfield instanceof Animal)); //true
        //System.out.println("¿Garfield es un perro? " + (garfield instanceof Perro)); Error de compilación

        System.out.println("¿Garfield es un gato? " + (garfield instanceof Gato)); //true
        System.out.println("¿Garfield es un felino? " + (garfield instanceof Felino)); //true
    }
}

```



```

<terminated> OperadorInstanceOf [Java Application] /Librar
¿Simba es un animal? true
¿Simba es un perro? false
¿Simba es un felino? false

¿Idex es un animal? true
¿Idex es un perro? true
¿Idex es un felino? false

¿Garfield es un animal? true
¿Garfield es un gato? true
¿Garfield es un felino? true

```

3. Clase abstracta

Una clase abstracta es aquella en la que alguno de sus métodos (al menos uno), están definidos como abstractos, es decir, está marcado como *abstract* y no está implementado, solo tiene su signatura.

Para declarar una clase como abstracta simplemente se debe poner la cláusula *abstract* y debe tener al menos un método abstracto:

```

public abstract class FactoryDAO {
    public abstract DAO getDAO();
}

```

Una clase abstracta no se puede instanciar, por tanto, dependen de clases derivadas para que se puedan utilizar sus métodos y atributos.

Las clases abstractas se suelen utilizar para definir una estructura y evitar repetir código, ya que sus métodos y atributos corresponden con las características y comportamientos comunes de sus clases hijas.

Las clases hijas que no sean abstractas deberán implementar todos los métodos abstractos.

En UML las clases abstractas aparece con el nombre en cursiva. Los métodos abstractos también aparecerán en cursiva:

<i>Vehiculo</i>
+velocidad:int=0
+ <i>acelera()</i> + <i>para()</i>

```
public abstract class Vehiculo {
    public int velocidad = 0;
    abstract public void acelera();
    ➤ public void para() {
        velocidad = 0;
    }
}

public class Coche extends Vehiculo {
    private int litros;
    protected String marca;

    ➤ public Coche(String marca) {
        this.marca = marca;
    }

    ➤ public void repostar(int litros) {
        this.litros = litros;
    }

    ➤ @Override
    public void acelera() {
        // Este método obligatoriamente debe estar definido
        velocidad+=5;
    }
}
```

4. Clases finales

Son clases definidas con la palabra reservada “final”. Este tipo de clases puede ser instanciada pero no utilizadas como base de herencia. Un ejemplo es la clase *String*, que puede ser utilizada, pero no la podemos tener de base para otras clases. En la siguiente imagen se puede ver una captura de la documentación oficial, sobre su definición:

```

java.lang
Class String

java.lang.Object
java.lang.String

All Implemented Interfaces:
    Serializable, CharSequence, Comparable<String>

public final class String
    extends Object
    implements Serializable, Comparable<String>, CharSequence

The String class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

    String str = "abc";

is equivalent to:

    char data[] = {'a', 'b', 'c'};
    String str = new String(data);

Here are some more examples of how strings can be used:

    System.out.println("abc");
    String cde = "cde";
    System.out.println("abc" + cde);
    String c = "abc".substring(2,3);
    String d = cde.substring(1, 2);

```

Por supuesto, la palabra reservada “*final*” es completamente contradictoria con “*abstract*”, es decir, no pueden existir clases finales y abstractas a la vez, es incoherente.

5. Interfaces

Una interfaz es una clase en la que ninguno de sus métodos está implementado, solo está su signatura.

Para la definición de una interfaz se utiliza la palabra reservada “*interface*”. Para implementar una interfaz se utiliza la palabra “*implements*”.

```

public interface FactoryDAO {
    public DAO getDAO();
}

public class FactoryOracle implements FactoryDAO{
    @Override
    public DAO getDAO() {
        // TODO Auto-generated method stub
        return null;
    }
}

```

Todas las clases derivadas deben implementar todos sus métodos.

Los métodos de una interfaz abstracta no deben llevar la palabra reservada “*abstract*”, sino que deberán ser simplemente “*public*”.

```

public interface Lista<T extends Comparable<? super T>> {
    /** Comprueba si la lista está vacía.
     * @return true si está vacía.
     */
    public boolean isEmpty();

    /** Obtiene el primer elemento.
     * @return el valor del primer nodo de la lista.
     */
    public T getFirst();

    /** Obtiene el último elemento.
     * @return el valor del último nodo de la lista.
     */
    public T getLast();

    /** Inserta un elemento al comienzo de la lista.
     * @param key valor del elemento.
     */
    public void insertAtBegin(T key);

    /** Inserta un elemento al final de la lista.
     * @param key valor del elemento.
     */
    public void insertAtEnd(T key);
}

```

Se considera buenas prácticas de programación el uso de interfaces ya que desacopla la aplicación y permite modificarla sin demasiados cambios.

5.1. Interfaz vs clase abstracta

Aunque son conceptos que en un principio se pueden parecer, hay varias distinciones que son importantes:

- Una clase solo puede extender de una clase abstracta, sin embargo podría implementar varias interfaces.
- Una clase abstracta puede implementar alguno de sus métodos, en el caso de las interfaces, ninguno de sus métodos debe estar implementado.
- En una clase abstracta, los métodos abstractos pueden ser *public* o *protected*. En una interfaz solamente puede haber métodos públicos.
- En una clase abstracta pueden existir variables *static*, *final* o *static final* con cualquier modificador de acceso (*public*, *private*, *protected* o *default*). En una interfaz sólo puedes tener constantes (*public static final*).

6. Clases internas (Inner Class)

Se trata de clases que se definen dentro de otra clase. Su uso no es nada recomendable, aunque algunos *frameworks*, como *swing* tienden a crearlas.

Hay una serie de consideraciones importantes:

- Para crear una instancia de la clase interna, será necesario disponer previamente de una instancia de la clase externa.

- Desde la clase interna tenemos acceso a los métodos y atributos privados de la clase externa.
- La instanciación de la clase externa se hace de forma habitual.

```

public class Externa {
    private int varExterna=150;

    public int duplicarVariable(){
        return varExterna*=2;
    }

    class Interna{
        public void metodoInterno(){
            System.out.println("Ini: "+varExterna);
            System.out.println("Dup: "+duplicarVariable());
        }
    }
}

public class ClaseInicio {
    public static void main(String[] args) {
        Externa ext = new Externa();
        System.out.println("DUP: "+ext.duplicarVariable());

        //Instancia de la interna
        Externa.Interna inter = ext.new Interna();
        inter.metodoInterno();
    }
}

```

Desde la clase interna accede a propiedades y métodos de la externa

Forma de acceder a la clase Interna

Observar que el operador new va precedido del punto y de una instancia de la clase externa.

Desde una clase interna se puede acceder a los miembros de una clase externa:

```

public class Externa {
    private int varExterna=150;
    public int duplicarVariable(){ return varExterna*=2;}

    public void metodoExterno(){
        System.out.println("Metodo Externo");
    }

    class Interna {
        public void metodoInterno(){
            System.out.println("Ini: "+varExterna);
            System.out.println(Externa.this.duplicarVariable());
        }

        public void metodoInterno_2(){
            this.metodoInterno();
            Externa.this.metodoExterno();
        }
    }
}

```

Si queremos hacer referencia desde la clase interna a la instancia de la clase externa

Existe varias variantes de este tipo de clases, como por ejemplo las clases internas locales a un método (*Method-local inner classes*), aunque su uso tiene bastantes limitaciones. Un ejemplo sería:

```
public class Externa {
    public void metodoExterno(){
        class Interna {
            public void metodoInterno(){
                System.out.println("LOCAL AL METODO");
            }
        }//Fin CI

        //A partir de aqui SI puede ser instanciada la clase interna
        Interna in = new Interna();
        in.metodoInterno();
    }// Fin METODO

    //A partir de qqui no puede ser instanciada la Clase Interna

    public static void main(String[] args) {
        new Externa().metodoExterno();
    }
}
```