

M03 Programació

Queralt, Josep

Revisió 3.00

Juny 30, 2021

Sumari

I. UF1 Programació estructurada	1
1. Resultats d'aprenentatge i criteris d'avaluació	7
2. Continguts	9
3. Breu història de Java	11
4. Les parts d'un programa Java	15
4.1. Compilar el codi font	16
4.2. Executar el codi compilat	20
4.3. Tipus d'errors	20
4.3.1. Errors en temps de compilació ("compile-time errors")	21
4.3.2. Errors en temps d'execució ("runtime errors")	22
4.3.3. Errors lògics	22
5. Mostrar literals, nombres i caràcters	23
5.1. Mostrar literals	23
5.2. Seqüències d'escapament	24
5.3. Mostrar caràcters	27
5.4. Mostrar nombres	28
6. Comentaris i Llegibilitat	31
7. Variables i tipus de dades	33
7.1. Què és una variable	33
7.2. Què és un identificador	33
7.3. Què és un tipus de dades	35
7.4. Diferències entre tipus primitius i tipus per referència	35
7.5. La gestió de la memòria en Java (el "stack" vs el "heap")	36
7.5.1. El "stack"	36
7.5.2. El "heap"	36
7.5.3. Quina de les estructures utilitzar en Java?	37
7.5.4. En resum	37
7.6. Declaració de variables	38
7.7. Assignació de variables	39
7.8. Assignació de variables pels tipus primitius	39
7.9. Assignació de variables pels tipus per referència	40
7.10. Tipus primitius en Java	40
7.10.1. Tipus Enters	41

7.10.2. Tipus int	41
7.10.3. Tipus long	43
7.10.4. Tipus byte	44
7.10.5. Tipus short	44
7.10.6. Tipus char	44
7.10.7. Tipus boolean	45
7.10.8. Tipus en coma flotant	45
7.10.9. Tipus float	48
7.10.10. Tipus double	48
8. Operadors	51
8.1. Operador d'assignació	52
8.2. Operadors aritmètics	53
8.3. Particularitats de l'operador /	55
8.4. Errors d'arrodoniment	55
8.5. Jerarquia dels operadors	56
8.6. Operador de concatenació de cadenes	57
8.7. Conversions de tipus entre dades numèriques (Castings)	58
8.8. Conversió implícita de tipus	59
8.9. Conversió explícita de tipus	59
9. Constants	61
9.1. Constants amb nom	61
10. System.in, System.out i System.err	63
11. Entrada i sortida de dades per la Consola	67
11.1. Sortida per pantalla amb format	67
11.2. Entrada de dades, classe Scanner	69
11.3. Problemes amb la classe Scanner	72
12. Programació estructurada	73
12.1. Orígens de la programació estructurada	73
12.1.1. Estructures de control derivades	74
12.2. Algorismes	76
12.3. Mitjans d'expressió d'un algorisme	76
12.4. Representació d'algorismes	77
12.4.1. Pseudocodi	77
12.5. Diagrama de control de flux	78
12.5.1. Alguns dels símbols utilitzats	79
12.6. Alguns algorismes d'exemple	80

12.6.1. Exemple 1	80
12.6.2. Exemple 2	80
12.6.3. Exemple 3	81
13. Estructures de selecció	83
13.1. Operadors relacionals	83
13.2. Estructura if	84
13.3. Estructura if-else	85
13.4. Estructures if-else aniuades	87
13.5. Estructures if-else encadenades	88
13.6. Variables de tipus interruptor "flag variables"	90
13.7. Operadors lògics	90
13.8. Estructura switch	92
13.9. Instrucció break	94
13.10. A vegades no utilitzar el break és útil	94
13.11. Operador ternari ?:	95
14. Estructures de repetició	97
14.1. Estructura while	97
14.2. Estructura do-while	99
14.3. Estructura for	101
14.4. Bucle "aniuats"	103
14.5. Instruccions break i continue	103
14.6. Control de dades d'entrada amb la classe Scanner	104
15. Tipus de dades Compostes	107
15.1. Matrius	107
15.1.1. Declaració de variables de tipus Matriu	107
15.1.2. Assignació i lectura de valors d' una matriu	109
15.1.3. Bucle for-each	111
15.1.4. Longitud d'una matriu, propietat length	111
15.2. Operacions habituals amb les matrius	111
15.2.1. Cerca d'un element dins d'una matriu	112
15.2.2. Clonar una matriu	112
15.2.3. Canvi de mida	114
15.2.4. Ordenació dels elements de la matriu	114
15.2.5. Comprovar si dues matrius són iguals	116
15.3. Vectors de vectors	117
15.3.1. Matrius multidimensionals	119

15.3.2. Declarar matrius de dues dimensions	119
15.3.3. Creació de matrius bidimensionals	120
15.4. Registres	122
15.4.1. Declarar l'estructura del registre	122
15.4.2. Crear nous registres a partir de l'estructura del registre	123
15.4.3. Valors per defecte	125
Bibliografia	127
II. UF2 Disseny modular	129
16. Resultats d'aprenentatge i criteris d'avaluació	135
17. Continguts	137
18. Tractament de cadenes	139
18.1. El tipus de dades String	139
18.1.1. Declaració i inicialització d'un objecte	140
18.1.2. Declaració i inicialització d'un objecte String	140
18.1.3. Els Strings són immutables	141
18.1.4. Combinar assignacions i sumes de cadenes pot ser molt ineficient	141
18.1.5. Els Strings no són adequats per fer experiments amb les referències	142
18.2. Mètodes i propietats de la classe String	142
18.3. Com s'interpreta una classe en UML	142
18.4. Mètodes de la classe String	145
18.4.1. equals(s1 : String): boolean	145
18.4.2. equalsIgnoreCase(s1: String): boolean	146
18.4.3. compareTo(s1: String): int	147
18.4.4. compareToIgnoreCase(s1: String): int	147
18.4.5. length(): int	148
18.4.6. charAt(index: int): char	148
18.4.7. substring(beginIndex: int): String	148
18.4.8. substring(beginIndex: int, endIndex: int): String	149
18.4.9. toLowerCase(): String	149
18.4.10. toUpperCase(): String	149
18.4.11. trim(): String	149
18.4.12. replace(oldChar: char, newChar: char): String	150

18.4.13. replaceFirst(oldString: String, newString: String): String	150
18.4.14. replaceAll(oldString: String, newString: String): String	150
18.4.15. split(delimiter: String): String[]	151
18.4.16. indexOf(ch: char): int	151
18.4.17. indexOf(ch: char, fromIndex: int): int	151
18.4.18. indexOf(s: String): int	151
18.4.19. indexOf(s: String, fromIndex: int): int	151
18.4.20. lastIndexOf(ch: int): int	152
18.4.21. lastIndexOf(ch: int, fromIndex: int): int	152
18.4.22. lastIndexOf(s: String): int	152
18.4.23. lastIndexOf(s: String, fromIndex: int): int	152
18.4.24. format(format, item1, item2, ..., itemk): String	152
18.5. Conversió entre cadenes i altres tipus de dades	152
18.5.1. toCharArray(): char[]	153
18.5.2. valueOf(...): String	153
18.6. Classes embolcall o Wrapper Classes	154
18.7. Conversió entre cadenes i números	155
18.8. El tipus de dades Character	155
18.9. Mètodes de la classe Character	156
18.10. Lectura de cadenes amb la classe Scanner	157
18.11. Cadenes i la configuració regional	158
18.12. Classes StringBuilder i StringBuffer	159
19. Mètodes (funcions)	161
19.1. Definir un mètode	162
19.2. Paraula clau return	163
19.3. Cridar un mètode	164
19.4. Ubicació dels mètodes	165
19.5. La pila de crides a funcions	166
19.6. Passar arguments per valor	168
19.7. Referències passades per valor	172
19.8. Passar matrius als mètodes	173
19.9. Tornar una matriu des d'un mètode	177
19.10. Arguments per la línia de comandes	177
19.11. Sobrecàrrega de mètodes	178

19.12. Àmbit de les variables	180
19.12.1. Variables d'àmbit de classe	180
19.12.2. Constants d'àmbit de classe	181
19.12.3. Variables d'àmbit de mètode	181
19.12.4. Variables d'àmbit de bloc	182
19.12.5. "shadowing"	183
19.13. Llista d'arguments de longitud variable	184
20. Programació modular en Java	187
20.1. Mètodes / funcions	187
20.2. Classes estàtiques	188
20.3. Biblioteques	189
20.3.1. Declaració de mòduls	189
20.3.2. Declaració de paquets	189
20.3.3. El paquet per defecte	191
20.4. Ubicació de les classes en paquets	192
20.4.1. Com ho fa Java per trobar les classes definides per l'usuari	192
20.4.2. Declaracions import	195
20.5. Fitxers jar	196
20.6. Les biblioteques del llenguatge java	198
20.6.1. Biblioteques abans i després de Java8	198
20.6.2. Moduls de Java11	198
20.6.3. Biblioteques de Java8	199
20.6.4. java.lang	199
20.6.5. java.util	200
20.6.6. java.util.Arrays	200
20.6.7. java.util.ArrayList	202
20.6.8. java.time	204
20.6.9. java.io	205
20.6.10. java.net	205
20.6.11. java.security	205
20.6.12. java.sql	205
20.6.13. java.swing	205
20.6.14. javafx	205
21. Alguns problemes difícils	207
21.1. Implementar l'eina "rellenar"	207

21.2. Realitzar cerques en una estructura de directoris	210
22. Recursivitat	213
22.1. Implementació interna en un ordinador	213
22.2. Algoritmes recursius	214
22.3. Exemple - comptaEnrrera	214
22.4. Exemple - mostrarLINES	215
22.5. Exemple - Càlcul del factorial d'un número	215
22.6. Exemple - Successió de Fibonacci	216
22.7. Exemple - Palindroms	217
22.8. Exemple - cercaBinaria	218
22.9. Conveniència de l'ús de la recursividat	219
22.10. Backtracking	219
23. Introducció a Processing	221
23.1. Mètode setup()	221
23.2. Mètode draw()	221
23.3. Paràmetres típics d'inicialització	221
23.4. Coordenades	222
23.5. Formes bàsiques	222
23.6. Mètode keyPressed()	223
23.7. Mètode mousePressed()	224
23.8. Mostrar text	225
23.9. Mostrar imatges	226
23.10. Temps	228
23.11. Animacions	229
23.12. Control de col·lisions	230
Bibliografia	233
III. UF3 Fonaments de gestió de fitxers	235
24. Resultats d'aprenentatge i criteris d'avaluació	239
25. Continguts	241
26. Gestió de fitxers	243
26.1. Entrada / Sortida	243
26.2. Paquets java.io i java.nio	243
26.3. Conèixer el directori de treball	244
26.4. "Paths" o rutes	245
26.5. Tractament d'errors en l'accés a fitxers	246
26.6. La classe File	247

26.7. Creació d'un objecte File	251
26.8. Comprovar si el fitxer existeix	252
26.9. Comprovar si el fitxer és un directori	252
26.10. Ruta absoluta i ruta canònica	252
26.11. Crear, renombrar i eliminar fitxers	254
26.12. Treballar amb els atributs dels fitxers	256
26.13. Copiar un fitxer	257
26.14. Saber la mida d'un fitxer	257
26.15. Mostrar tots els directoris arrel	257
26.16. Mostrar tots els fitxers i directoris d'un directori	258
27. Lectura i escriptura de fitxers	261
27.1. Classes de la biblioteca de Java	262
27.2. Per què calen dos conjunts de classes?	262
27.3. Tractament seqüencial o aleatori	263
27.4. Java IO: Streams	263
27.5. Java IO: Readers i Writers	264
28. Errors a la lectura/escriptura de fitxers	267
29. Lectura i escriptura de fitxers binaris d'accés seqüencial	269
29.1. Classe java.io.FileInputStream i FileOutputStream	269
29.2. Escriure un fitxer Byte a Byte	269
29.3. Sobreescriure el fitxer o afegir dades al final	271
29.4. Llegir un fitxer Byte a Byte	271
29.5. Mètodes de la classe FileInputStream	273
29.5.1. Un Exemple	274
29.5.2. Un altre exemple	275
29.6. Mètodes de la classe OutputStream	275
29.6.1. Un exemple	276
29.7. Exemple Xifrat del cèsar per bytes	277
30. Lectura i escriptura de fitxers de text a baix nivell	279
30.1. Caràcters i Unicode	279
30.2. Classe java.io.OutputStreamWriter	280
30.2.1. Determinar la codificació de caràcters	281
30.2.2. Tancar un OutputStreamWriter	281
30.3. Sobreescriure el fitxer o afegir dades al final	281
30.4. Classe java.io.InputStreamReader	282
30.4.1. Mètode read()	283

30.4.2. Final de fitxer	283
30.5. Selecció de la codificació de caràcters	284
30.5.1. Tancar un InputStreamReader	284
30.6. Classe java.io.FileReader i java.io.FileWriter	284
30.7. Sobreescriure el fitxer o afegir dades al final	285
31. Lectura i escriptura de fitxers de text a alt nivell	287
31.1. Classe Scanner	287
31.2. Exemple lectura d'un fitxer de text mitjançant la classe Scanner	288
31.3. Mètodes de la classe java.util.Scanner	290
31.4. Classe java.io.PrintWriter	291
31.5. Utilització d'un objecte PrintWriter	291
31.6. Mètodes de la classe java.io.PrintWriter	293
31.7. Exemple d'utilització de PrintWriter	294
31.8. Llegir un fitxer de text per paraules	295
31.9. Classe Scanner i la codificació de caràcters	298
31.10. Llegir un fitxer de text caràcter a caràcter	298
31.11. Classificar caràcters	298
31.12. Llegir un fitxer de text línia a línia	298
31.13. Escanejar una cadena	298
Bibliografia	301
IV. UF4 Programació orientada a objectes. Fonaments	303
32. Resultats d'aprenentatge i criteris d'avaluació	309
33. Continguts	311
34. Classes i Objectes	313
34.1. Què és una classe i què és un objecte?	314
34.2. Declarar una classe	315
34.3. Les classes defineixen nous tipus de dades per referència ..	315
34.4. Crear instàncies d'una classe	316
34.5. Declarar camps en una classe	316
34.6. El tipus null	318
34.7. L'operador . per accedir als camps d'una classe	319
34.8. Inicialització per defecte dels camps	320
34.9. Revisió de la gestió de la memòria en Java	320
34.9.1. Quina de les estructures utilitzar en Java?	322
34.10. Tots les classes són tipus per referència	322

34.11. L'operador d'assignació i els tipus per referència	322
34.11.1. Assignació i tipus primitius	322
34.11.2. Assignació i tipus per referència	323
34.12. Modificadors d'accés per una classe	324
34.13. Revisió de la clàusula import	326
34.14. Declaració de mètodes en una classe	327
34.15. Variables locals	327
34.16. Mètodes d'una instància i mètodes d'una classe	328
34.17. Invocar un mètode	329
34.18. Constants	330
34.19. Revisió del mètode main	331
34.20. Paraula clau this	331
34.21. Modificadors d'accés pels membres d'una classe	333
34.22. Representació de classes en UML	333
34.23. Primer principi de la programació orientada a objectes, l'encapsulació i l'amagament de la informació	335
34.24. L'encapsulació a la pràctica	337
34.25. Constructors	338
34.26. Sobrecàrrega de constructors	340
34.27. Cridar un constructor des d'un altre constructor	341
34.28. Constructor per defecte	343
34.29. Representació de les relacions entre classes en UML	344
34.30. Representació de les relacions entre classes en UML - Exemple	347
34.31. Exemples	347
34.31.1. CercleSimple	347
34.31.2. TV	349
35. Principis de disseny de classes	353
35.1. Principi de Cohesió	353
35.2. Principi de Consistència	353
35.3. Principi d'encapsulació	354
35.4. Principi de Claredat	354
36. La classe Object (Primera part)	355
36.1. Totes les instàncies són un objecte	355
36.2. Els mètodes de la classe Object	355
36.3. Quina és la classe d'un objecte?	357

36.4. Comparar objectes per saber si són iguals	358
36.4.1. Especificacions alhora de reimplementar el mètode equals()	360
36.5. Calcular el "hash code" d'un objecte	361
36.5.1. Per què els objectes mantenen un "hash" code?	361
36.5.2. Quan cal reimplementar el mètode <i>hashCode()</i>	362
36.5.3. Exemple	362
36.6. Representació d'un objecte amb una cadena	364
36.7. Exemple - Una classe completa	365
36.8. Finalitzar un Objecte - Deprecated JDK11	367
37. Herència	369
37.1. Superclasses i subclasses	369
37.2. Representació de la relació d'herència en UML	370
37.3. Paraula clau extends	371
37.4. La classe Object és la superclasse per defecte	371
37.5. Herència i la relació jeràrquica	372
37.6. Quines coses són heretades per una subclasse?	373
37.7. Herència múltiple	374
37.8. Upcasting i Downcasting	375
37.8.1. Upcasting	375
37.8.2. Downcasting	375
37.9. Operador instanceof	376
37.10. Compte amb l'operador instanceof i el mètode equals ...	377
37.11. Sobreescriptura de mètodes	379
37.12. Anotació @override	382
37.13. Cridar un mètode sobreescrit des d'una classe filla, paraula clau super	382
37.14. Upcasting i sobreescriptura, "overriding"	383
37.15. Les variables en Java no es poden sobreescriure	384
37.16. Accés a mètodes sobreescrits	386
37.17. Sobrecàrrega de mètodes, "overloading"	387
37.18. L'herència i els constructors	387
37.19. Amagar mètodes	393
37.20. Amagar camps	393
37.21. Deshabilitar l'herència	393
37.22. Paraula clau final	394

37.22.1. Variables final	394
37.22.2. Mètodes final	394
37.22.3. Classes final	394
37.23. Objectes immutables	396
37.23.1. Quins avantatges tenen els objectes immutables ..	397
37.23.2. Estratègies per a definir objectes immutables ..	397
37.24. Classes i mètodes abstractes	402
37.25. Composició per sobre de l'herència	404
38. Interfícies	407
38.1. Tenim un problema....	407
38.2. Una solució mitjançant interfícies	412
38.3. Declarar una interfície	414
38.4. Mètodes estàtics a les interfícies	415
38.5. Mètodes per defecte a les interfícies	415
38.6. Declaració d'interfícies niuades	415
38.7. Una interfície defineix un nou tipus de dades	415
38.8. Implementar una interfície	415
38.9. Interfícies en UML	416
38.10. Implementar múltiples interfícies	417
38.11. Implementació parcial d'interfícies	418
38.12. Les interfícies admeten herència (i herència múltiple) ..	418
38.13. El Polimorfisme revisat	419
39. Principis de disseny a la programació orientada a objectes	421
39.1. Hem de sospitar que el nostre codi té problemes si....	421
39.2. Príncipi de responsabilitat única	421
39.3. Príncipi d'obert-tancat	422
39.4. Príncipi de substitució de Liskov	424
39.5. Príncipi de la segregació d'interfícies	425
39.6. Príncipi de la inversió de les dependències	426
V. UF5 POO. Llibreries de classes fonamentals	429
40. Resultats d'aprenentatge i criteris d'avaluació	441
41. Continguts	443
42. Control d'excepcions	445
42.1. Llançament d'excepcions	445
42.2. Excepcions	447
42.3. Com es gestionen les "checked exception"	448

42.4. Com es gestionen les RuntimeException	449
42.5. Captura d' excepcions	450
42.6. Obtenir informació de les excepcions	452
42.7. Clàusula finally	453
42.8. Definir noves classes d' excepció	454
42.9. Construcció try-amb-recursos	456
42.10. Algunes recomanacions a l' hora de treballar amb excepcions	458
42.10.1. Llençar d' hora capturar tard	458
42.10.2. No emmascarar excepcions	459
42.10.3. Tenir en compte que algunes excepcions es poden produir al bloc finally	459
43. La classe ArrayList	463
43.1. <code>java.util.ArrayList</code> versió anterior a Java 5	463
43.2. <code>java.util.ArrayList</code> versió posterior a Java 5	469
44. Genèrics	471
44.1. Tipus genèrics	473
44.2. Implementació de la classe Box	473
44.3. La classe Box utilitzant genèrics	474
44.4. Noms de les <i>variables tipus</i>	475
44.5. Referències a classes genèriques	475
44.6. Classes genèriques amb més d'un paràmetre	475
44.6.1. Utilitzar classes genèriques sense <i>arguments tipus</i> ..	476
44.7. Tipus acotats (bounded types)	477
44.7.1. Múltiples tipus acotats	479
44.8. Arguments comodí	479
44.8.1. Primer intent de solució	480
44.8.2. Segon intent de solució	481
44.8.3. Tercer intent de solució	482
44.8.4. Codi d'exemple	482
44.9. Arguments comodí acotats	483
44.9.1. Codi d'exemple	487
44.10. Mètodes genèrics	490
44.11. Constructors genèrics	491
44.12. Interfícies genèriques	492
45. La interfície Comparable	495

46. La interfície Comparator	503
46.1. Comparator i Comparable per les mateixes classes	505
47. Col·leccions	507
47.1. Interfícies vs implementacions	507
47.2. Programar per la interfície, no per la implementació	508
47.3. Un exemple, una interfície, dues implementacions	509
47.3.1. La interfície	510
47.3.2. Les dues implementacions	510
47.3.3. Les proves	513
47.4. Interfícies	514
47.4.1. La interfície <code>java.util.Collection</code>	515
47.4.2. Interfície <code>java.util.Queue</code> (Cua)	516
47.4.3. Interfície <code>java.util.Deque</code>	517
47.4.4. Interfície <code>java.util.List</code> (Llista)	518
47.4.5. Interfície <code>java.util.Set</code> (Conjunt)	519
47.4.6. Interfície <code>java.util.Map</code> (Diccionari)	520
47.5. Implementacions	521
47.5.1. Implementacions de <code>java.util.Queue</code>	521
47.5.2. Implementacions de <code>java.util.Deque</code>	522
47.5.3. Implementacions de <code>java.util.List</code>	523
47.5.4. Implementacions de <code>java.util.Set</code>	523
47.5.5. Implementacions de <code>java.util.Map</code>	524
47.6. Recórrer col·leccions	527
47.7. Iteradors	527
47.8. Exemple d'implementació d'un iterador	532
47.8.1. Els iteradors de llistes: <code>ListIterator</code>	536
47.8.2. Inserir i extreure elements a través d'un iterador	536
47.8.3. Ús d'un iterador sense l'iterador: <code>enhanced for</code>	537
47.9. Algorismes: les classes <code>Arrays</code> i <code>Collections</code>	539
47.10. Configuració de JavaFX i Apache Netbeans amb Ant	541
47.10.1. Baixar i instal·lar la biblioteca	541
47.10.2. Afegir la biblioteca JavaFX globalment	541
47.10.3. Afegir JavaFX a un projecte concret	544
47.11. Configuració de JavaFX i Apache Netbeans amb Maven ..	547
48. JavaFX - Introducció	549
48.1. Introducció a les interfícies gràfiques en Java	549

48.2. Estructura bàsica d'un programa de JavaFX	550
48.3. Aplicacions amb més d'un Stage	552
48.4. Panells, Controls i Formes	553
48.5. Com s'utilitza un Pane amb un Control	555
48.6. La classe Pane	556
48.7. Com s'utilitza un Pane amb una Shape	557
48.8. Classe Color	559
48.9. Classe Font	559
48.10. Aplicar fulls d'estil CSS	561
48.10.1. Aplicar un estil directament	563
48.10.2. Crear un full d'estils i aplicar-lo a la escena	564
48.11. Classe Image i ImageView	568
48.12. Layouts	570
48.12.1. FlowPane	571
48.12.2. TextFlow	573
48.12.3. TilePane	576
48.12.4. GridPane	579
48.12.5. BorderPane	581
48.12.6. HBox i VBox	583
48.12.7. AnchorPane	586
48.13. Creació de nous controls	587
48.13.1. Crear un nou control per composició	588
48.13.2. Crear un nou control per herència	588
49. Inner classes i Nested classes	591
49.1. Què és una "nested class"	591
49.2. Representació d'una classe "nested" en UML	592
49.3. Per què necessitem les nested class?	592
49.4. Tipus de "inner classes"	593
49.4.1. "inner class" com a membre d'una altra classe	593
49.4.2. Classe "static nested class"	594
49.4.3. "inner class" local	595
49.4.4. "inner class" anònima	596
49.5. Exemple 1 - Inner class	597
49.6. Exemple 2 - Local class	598
49.7. Exemple 3 - Classe anònima	599
50. Interfícies funcionals	601

50.1. Com encapsular una funció en una variable	601
50.2. Proposta solució - versió 1	601
50.3. Proposta solució - versió 2	602
50.4. Proposta solució - versió 3	603
50.5. Interfícies funcionals	603
50.6. Mètodes default a les interfícies funcionals	604
50.7. Mètodes static a les interfícies funcionals	605
51. Expressions Lambda	607
51.1. Estructura de la llista d'arguments	608
51.2. Estructura del cos de la expressió lambda	609
51.3. Abast d'una expressió Lambda	610
51.4. Predicats	611
51.5. Exemples	612
51.5.1. Exemple 1	612
51.5.2. Exemple 2	612
51.5.3. Exemple 3	612
51.5.4. Exemple 4	613
51.5.5. Exemple 5	613
52. Exemple - Classes anònimes i Lambdes	615
52.1. Aproximació 1: Crear mètodes que busquin membres per una característica	618
52.2. Aproximació 2: Crear mètodes de cerca més generals	619
52.3. Aproximació 3: Especificar el codi del criteri de cerca en una classe local	619
52.4. Aproximació 4: Especificar el codi del criteri de cerca en una interfície	620
52.5. Aproximació 5: Especificar el codi del criteri de cerca en una classe anònima	621
52.6. Aproximació 6: Especificar el codi del criteri de cerca en una expressió Lambda	621
52.7. Aproximació 7: Utilitzar interfícies funcionals estàndards i expressions Lambda	621
52.8. Aproximació 8: Utilitzar expressions Lambda a tota l'aplicació	622
52.9. Aproximació 9: Introduir genèrics	623
53. Fluxos	625

53.1.	Iteracions vs Streams	626
53.1.1.	Exemple - Cerca de productes versió Col·leccions ..	626
53.1.2.	Exemple - Cerca de productes versió Streams	629
53.2.	Creació de fluxos	630
53.2.1.	Directament a partir dels seus elements	630
53.2.2.	A partir d'una matriu	630
53.2.3.	A partir d'una List	631
53.2.4.	A partir d'una funció generadora	631
53.3.	Convertir un Stream a una col·lecció	632
53.3.1.	Recuperar els elements en una List	632
53.3.2.	Recuperar els elements en una matriu	633
53.4.	Operacions intermèdies	633
53.4.1.	Stream.filter()	633
53.4.2.	Stream.map()	634
53.4.3.	Stream.sorted()	635
53.5.	Operacions terminals	635
53.5.1.	Stream.forEach()	636
53.5.2.	Stream.collect()	636
53.5.3.	Stream.match()	637
53.5.4.	Stream.count()	638
53.5.5.	Stream.reduce()	638
53.6.	Operacions interruptibles	639
53.6.1.	Stream.anyMatch ()	639
53.6.2.	Stream.findFirst()	640
53.7.	Streams paral·lels	641
54.	JavaFX - Esdeveniments	643
54.1.	Esdeveniments	643
54.1.1.	Creant una classe que implementa EventHandler<ActionEvent>	645
54.1.2.	Utilitzant una classe anònima	646
54.1.3.	Utilitzant una expressió lambda	647
54.1.4.	Centralitzant la gestió d'esdeveniments en un sol mètode	648
54.2.	Alguns esdeveniments comuns	649
54.2.1.	Esdeveniments del ratolí	649
54.2.2.	Esdeveniments del teclat	650

55. Patró Model-vista-Controlador	653
55.1. Avantatges i problemes del model	654
55.1.1. Avantatges	654
55.1.2. Problemes	655
55.2. Esquelet del patró	655
55.2.1. El model	655
55.2.2. La vista	656
55.2.3. El controlador	656
55.2.4. El mètode main	657
55.3. Un exemple d'implementació	658
55.3.1. El model	658
55.3.2. La vista	659
55.3.3. El Controlador	662
55.3.4. El mètode main	663
56. Patró Observer	665
56.1. Definició del patró	665
56.2. Implementació en Java	665
57. Vincular atributs de dues classes	669
57.1. El codi complet	675
58. JavaFX - Binding	679
58.1. Les propietats de JavaFX	679
58.1.1. Vincular propietats	679
58.1.2. Implementació 1 - Utilitzant directament el patró observer	680
58.1.3. Implementació 2 - Utilitzant bindings	681
58.1.4. Explicació de l'exemple	683
58.1.5. Escoltar propietats	685
58.2. Interfície ObservableList	686
58.3. ObservableList i TableView	689
58.4. Exemple - TableView	690
59. JavaFX - FXML	697
59.1. Creació d'una vista	697
59.2. Carregar un fitxer FXML	698
59.3. Afegir un controlador	699
59.3.1. Crear la classe controlador	699
59.3.2. Posar nom als nodes de la vista	700

59.3.3. Crear variables al controlador pels nodes de la vista	701
59.3.4. Gestió d'esdeveniments	702
60. JavaFX FXML - Un exemple guiat	705
60.1. Implementació de la Vista	706
60.1.1. Paquets	706
60.1.2. Formulari ContactesMestre	706
60.1.3. Scene Builder	708
60.1.4. Disseny del formulari ContactesMestre	709
60.1.5. Creació del formulari principal	715
60.1.6. Implementació de la funció main	716
60.2. Implementació dels Model / Controlador	718
60.2.1. Model	718
60.2.2. L'origen de dades	720
60.2.3. ContacteController	722
60.2.4. Enllaçar l'aplicació principal amb ContactesMestreController	724
60.2.5. Vincular la vista al controlador	726
60.3. Esdeveniments	728
60.3.1. Resposta a canvis en la selecció de la Taula	728
60.3.2. Detecta canvis de selecció en la taula	729
60.3.3. El botó d'esborrar	730
60.3.4. Gestió d'errors	731
60.3.5. Diàlegs per a crear i editar contactes	732
60.3.6. Enllaçar la vista i el controlador	734
60.3.7. Obrint la finestra de diàleg	735
61. Entrada / Sortida	737
61.1. Paquets java.io i java.nio	737
61.2. Fluxos basats en bytes i fluxos basats en text	738
62. Lectura i escriptura de fitxers	741
62.1. Classes de la biblioteca de Java	742
62.2. Per què calen dos conjunts de classes?	742
62.3. Tractament seqüencial o aleatori	743
62.4. Java IO: Streams	743
62.5. Java IO: Readers i Writers	745
63. Treballar amb fitxers	749

63.1. Conèixer el directori de treball	749
63.2. "Paths" o rutes	750
63.3. La classe File	751
63.4. Creació d'un objecte File	755
63.5. Comprovar si el fitxer existeix	756
63.6. Comprovar si el fitxer és un directori	756
63.7. Ruta absoluta i ruta canònica	756
63.8. Crear, renombrar i eliminar fitxers	759
63.9. Treballar amb els atributs dels fitxers	760
63.10. Copiar un fitxer	761
63.11. Saber la mida d'un fitxer	761
63.12. Mostrar tots els directoris arrel	762
63.13. Mostrar tots els fitxers i directoris d'un directori	762
64. Lectura i escriptura de streams binaris d'accés seqüencial	765
64.1. Classe java.io.InputStream	765
64.1.1. Mètode read()	766
64.1.2. Final del stream	766
64.1.3. Mètode read(byte[])	766
64.1.4. Mètodes mark() i reset()	767
64.1.5. Exemple 1	768
64.1.6. Exemple 2	768
64.2. Exemple 3	769
64.3. Classe Java.io.OutputStream	770
64.3.1. Mètode write(int)	770
64.3.2. Mètode flush()	771
64.3.3. Mètode close()	771
64.4. Classe java.io.FileInputStream i FileOutputStream	771
64.5. Mètodes de la classe InputStream	773
64.6. Mètodes de la classe OutputStream	774
64.6.1. Exemple Xifrat del cèsar	775
65. Lectura i escriptura de fitxers de text a baix nivell	777
65.1. Tractament de possibles errors d'accés al fitxer	777
65.2. Classe java.io.Reader	778
65.2.1. Caràcters i Unicode	778
65.2.2. Llegir caràcters amb un Reader	779
65.3. Classe java.io.Writer	779

65.3.1. Caràcters i Unicode	780
65.4. Classe java.io.InputStreamReader	780
65.4.1. Mètode read()	782
65.4.2. Final de fitxer	782
65.4.3. Tancar un InputStreamReader	783
65.5. Classe java.io.OutputStreamWriter	783
65.5.1. Determinar la codificació de caràcters	784
65.5.2. Tancar un OutputStreamWriter	785
65.6. Classe java.io.FileReader	785
65.7. Classe java.io.FileWriter	786
65.7.1. Sobreescriure el fitxer o afegir dades al final	787
65.8. La classe java.io.BufferedReader	787
66. Lectura i escriptura de fitxers de text a alt nivell	789
66.1. Classe Scanner	789
66.2. Exemple lectura d'un fitxer de text mitjançant la classe Scanner	790
66.3. Mètodes de la classe Scanner	792
66.4. Classe java.io.PrintWriter	793
66.5. Utilització d'un objecte PrintWriter	793
66.6. Mètodes de la classe PrintWriter	795
66.7. Exemple d'utilització de PrintWriter	796
66.8. Llegir un fitxer de text per paraules	797
66.9. Classe Scanner i la codificació de caràcters	800
66.10. Llegir un fitxer de text caràcter a caràcter	800
66.11. Classificar caràcters	800
66.12. Llegir un fitxer de text línia a línia	800
66.13. Escanejar una cadena	800
66.14. Com obrir un PrintWriter per afegir dades	801
67. Lectura i escriptura en de fitxers binaris d'accés aleatori	803
67.1. Escriptura seqüencial sobre un fitxer aleatori	804
67.2. Classe RandomAccessFile	805
67.3. Mètodes de la classe RandomAccessFile	811
68. Expressions regulars	815
68.1. Sintaxi de les expressions regulars	815
68.2. Algunes expresions regulars d'exemple	817
68.3. Treballar amb expressions regulars	818

68.3.1. Aplicar expressions regulars sobre objectes String ..	818
68.3.2. Altres mètodes regex de la classe String	820
68.4. Classes específiques per les expressions regulars	820
68.4.1. La classe Pattern	821
68.4.2. La classe Matcher	821
68.5. Alguns exemples	822
68.5.1. Exemple 1	822
68.5.2. Exemple 2	823
68.5.3. Exemple 3	823
68.5.4. Exemple 4	824
Bibliografia	827
VI. UF6 POO. Introducció a la persistència en les BD	829
69. Resultats d'aprenentatge i criteris d'avaluació	833
70. Continguts	835
71. Introducció a JDBC	837
71.1. ODBC - Open DataBase Connectivity	837
71.2. JDBC - Java Database Connectivity	838
71.3. Drivers JDBC	839
71.3.1. Tipus de drivers JDBC	839
72. Instal·lació del MySQL/MariaDB	841
72.1. Instal·lació de MySQL/MariaDB en Debian	841
72.2. Instal·lació de MySQL/MariaDB en Windows	842
72.3. Importació de bases de dades d'exemple	842
72.4. Creació d'usuaris	843
72.4.1. Creació d'usuaris	843
72.4.2. Veure la llista d'usuaris creats	845
73. Establiment de connexions	847
73.1. Incorporació del driver al projecte	847
73.2. Prova de connexió	849
73.3. Formes alternatives d'establir la connexió	851
74. Sentències DML. Select	853
74.1. Obtenir els camps d'un registre a partir del nom del camp ..	853
74.2. Alguns tipus de dades poden presentar dificultats	854
74.3. Obtenir els camps d'un registre a partir de l'índex del camp ..	857
74.4. Seleccions parametritzades i injecció de codi	858
75. Sentències preparades	863

75.1. SQL-Injection	863
75.2. Ús de les sentències preparades	865
75.3. LIKE vs =	867
76. Sentències DML i DDL. Modificació de dades	869
76.1. Obtenció dels <i>id</i> autogenerats	871
77. Interfície Resultset	873
77.1. Tipus de resultsets	873
77.2. Concurrència d'un ResultSet	874
77.3. Mantenir el cursor en un ResultSet	875
77.4. Modificar dades a través d'un ResultSet	875
77.4.1. Modificar el valor d'una columna	875
77.4.2. Inserir un nou registre	876
77.4.3. Eliminar un registre existent	876
77.5. Un exemple sencer	877
78. Metadades	879
78.1. Mètode main	881
78.2. Obtenir informació de la base de dades	881
78.3. Obtenir informació de les taules de la base de dades	882
78.4. Obtenir informació d'una taula	884
78.5. Metadades de consultes	885
78.6. Sentències explain	887
79. Procediments emmagatzemats	891
79.1. Configuració de MariaDB	891
79.2. Crida a procediments	893
79.3. Crida a funcions	895
79.4. Crida a procediments des de Java	898
79.5. Crides a funcions des de Java	900
80. Transaccions	901
80.1. Autocommit	901
80.2. Exemple	902
Bibliografia	905
A. Manual d'estil	907
A.1. Avís important	908
A.2. Variables	909
A.2.1. Notació "Camel-case"	909
A.2.2. Constants	909

A.2.3. Números màgics	910
A.2.4. Utilitzar noms que mostrin les intencions	910
A.2.5. Evitar caràcters no americans als noms de les variables ...	911
A.3. Format	911
A.3.1. Sagnat	911
A.3.2. Format vertical	913
A.4. Control de flux	915
A.4.1. Bucles for	915
A.4.2. Ús de break i continue	916
A.4.3. Ús de System.exit()	916
A.5. Funcions i mètodes	916
A.5.1. Han de tenir un mida reduïda	916
A.5.2. Han de fer una sola cosa	916
A.5.3. Han de tenir noms descriptius	917
A.5.4. Han de tenir pocs paràmetres d'entrada	917
A.5.5. El nom dels paràmetres ha de ser explícit	917
A.6. Comentaris	917
A.6.1. Els comentaris no compensen el codi incorrecte	917
A.6.2. És millor que el codi sigui autoexplicatiu	918
A.6.3. Bons comentaris	918
A.6.4. Mals comentaris	918
A.7. Classes	919
A.7.1. Notació	919
A.7.2. Noms de classes	919
A.8. Excepcions	920
A.8.1. Excepcions genèriques	920
A.8.2. No amagar les excepcions	920
B. Sistemes de representació de la informació numèrica	923
B.1. Teorema fonamental dels nombres	925
B.2. Exemples de conversió a decimal	926
B.3. Conversió entre sistemes de numeració	926
B.4. Conversió d'una base b a base 10	926
B.5. Conversió de base decimal a base b	928
B.5.1. Per la part entera	928
B.5.2. Per la part decimal	928
B.5.3. Conversió de base b a base b'	929

B.6. Cas especial - Conversió de binari a Hexadecimal i viceversa	930
B.7. El sistema binari	931
B.8. Operacions bit a bit	933
B.9. Tipus d'operacions	933
B.10. Operacions bit a bit	934
B.10.1. NOT	934
B.10.2. AND	934
B.10.3. OR	935
B.10.4. XOR	935
B.10.5. Operacions de desplaçament	935
C. Representació de dades a la memòria de l'ordinador	937
C.1. Representació d'enters	938
C.2. Enters sense signe	938
C.3. Enters amb signe	939
C.3.1. Representació en signe-valor absolut	939
C.3.2. Representació en complement a 1	941
C.3.3. Representació en complement a 2	943
C.3.4. Descodificar nombres representats en complement a 2 ...	947
C.4. Big Endian vs. Little Endian	948
C.5. Representació de nombres en coma flotant	949
C.5.1. L'estàndard normalitzat IEEE 754	950
C.5.2. Forma de-normalitzada	953
C.6. Codificació de caràcters	954
C.6.1. ASCII	954
C.6.2. ISO 8859	956
C.6.3. UNICODE	956
C.6.4. UTF-8	957
C.6.5. Codificació del caràcter de final de línia	958
D. Generació de nombres aleatoris en Java	959
D.1. Nombres aleatoris	960
D.2. Classe Random	960
E. Diagrames de classes amb UML	963
E.1. Especificadors d'accés	964
E.2. Altres especificadors	964
E.3. Relacions entre classes	965
E.4. Misèl·lia	965

F. Operacions amb bits	967
F.1. Operacions amb bits	968
G. Llicència	971
G.1. Llicència pel text	971
G.2. Llicència pel codi	972
Index	973

Part I. UF1 Programació estructurada

Bad programming is easy. Idiots can learn it in 21 days, even if they are dummies.

— Matthias Felleisen *professor i autor en les ciències de la computació.*

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

— Martin Fowler *enginyer de software britànic especialitzat en anàlisi i disseny orientat a objectes.*

Computer science education cannot make anybody an expert programmer any more than studying brushes and pigment can make somebody an expert painter.

— Eric S. Raymond *Autor del llibre "la catedral i el basar" famós assaig sobre el software de codi obert.*

Sumari

1. Resultats d'aprenentatge i criteris d'avaluació	7
2. Continguts	9
3. Breu història de Java	11
4. Les parts d'un programa Java	15
4.1. Compilar el codi font	16
4.2. Executar el codi compilat	20
4.3. Tipus d'errors	20
4.3.1. Errors en temps de compilació ("compile-time errors")	21
4.3.2. Errors en temps d'execució ("runtime errors")	22
4.3.3. Errors lògics	22
5. Mostrar literals, nombres i caràcters	23
5.1. Mostrar literals	23
5.2. Seqüències d'escapament	24
5.3. Mostrar caràcters	27
5.4. Mostrar nombres	28
6. Comentaris i Llegibilitat	31
7. Variables i tipus de dades	33
7.1. Què és una variable	33
7.2. Què és un identificador	33
7.3. Què és un tipus de dades	35
7.4. Diferències entre tipus primitius i tipus per referència	35
7.5. La gestió de la memòria en Java (el "stack" vs el "heap")	36
7.5.1. El "stack"	36
7.5.2. El "heap"	36
7.5.3. Quina de les estructures utilitzar en Java?	37
7.5.4. En resum	37
7.6. Declaració de variables	38
7.7. Assignació de variables	39
7.8. Assignació de variables pels tipus primitius	39
7.9. Assignació de variables pels tipus per referència	40
7.10. Tipus primitius en Java	40
7.10.1. Tipus Enters	41
7.10.2. Tipus int	41
7.10.3. Tipus long	43

7.10.4. Tipus byte	44
7.10.5. Tipus short	44
7.10.6. Tipus char	44
7.10.7. Tipus boolean	45
7.10.8. Tipus en coma flotant	45
7.10.9. Tipus float	48
7.10.10. Tipus double	48
8. Operadors	51
8.1. Operador d'assignació	52
8.2. Operadors aritmètics	53
8.3. Particularitats de l'operador /	55
8.4. Errors d'arrodoniment	55
8.5. Jerarquia dels operadors	56
8.6. Operador de concatenació de cadenes	57
8.7. Conversions de tipus entre dades numèriques (Castings)	58
8.8. Conversió implícita de tipus	59
8.9. Conversió explícita de tipus	59
9. Constants	61
9.1. Constants amb nom	61
10. System.in, System.out i System.err	63
11. Entrada i sortida de dades per la Consola	67
11.1. Sortida per pantalla amb format	67
11.2. Entrada de dades, classe Scanner	69
11.3. Problemes amb la classe Scanner	72
12. Programació estructurada	73
12.1. Orígens de la programació estructurada	73
12.1.1. Estructures de control derivades	74
12.2. Algorismes	76
12.3. Mitjans d'expressió d'un algorisme	76
12.4. Representació d'algoritmes	77
12.4.1. Pseudocodi	77
12.5. Diagrama de control de flux	78
12.5.1. Alguns dels símbols utilitzats	79
12.6. Algunes algoritmes d'exemple	80
12.6.1. Exemple 1	80
12.6.2. Exemple 2	80

12.6.3. Exemple 3	81
13. Estructures de selecció	83
13.1. Operadors relacionals	83
13.2. Estructura if	84
13.3. Estructura if-else	85
13.4. Estructures if-else aniuades	87
13.5. Estructures if-else encadenades	88
13.6. Variables de tipus interruptor "flag variables"	90
13.7. Operadors lògics	90
13.8. Estructura switch	92
13.9. Instrucció break	94
13.10. A vegades no utilitzar el break és útil	94
13.11. Operador ternari ?:	95
14. Estructures de repetició	97
14.1. Estructura while	97
14.2. Estructura do-while	99
14.3. Estructura for	101
14.4. Bucles "aniuats"	103
14.5. Instruccions break i continue	103
14.6. Control de dades d'entrada amb la classe Scanner	104
15. Tipus de dades Compostes	107
15.1. Matrius	107
15.1.1. Declaració de variables de tipus Matriu	107
15.1.2. Assignació i lectura de valors d' una matriu	109
15.1.3. Bucles for-each	111
15.1.4. Longitud d'una matriu, propietat length	111
15.2. Operacions habituals amb les matrius	111
15.2.1. Cerca d'un element dins d'una matriu	112
15.2.2. Clonar una matriu	112
15.2.3. Canvi de mida	114
15.2.4. Ordenació dels elements de la matriu	114
15.2.5. Comprovar si dues matrius són iguals	116
15.3. Vectors de vectors	117
15.3.1. Matrius multidimensionals	119
15.3.2. Declarar matrius de dues dimensions	119
15.3.3. Creació de matrius bidimensionals	120

15.4. Registres	122
15.4.1. Declarar l'estructura del registre	122
15.4.2. Crear nous registres a partir de l'estructura del registre ...	123
15.4.3. Valors per defecte	125
Bibliografia	127

1

Resultats d'aprenentatge i criteris d'avaluació

1. Reconeix l'estructura d'un programa informàtic, identificant i relacionant els elements propis del llenguatge de programació utilitzat.
 1. Identifica els blocs que componen l'estructura d'un programa informàtic.
 2. Crea projectes de desenvolupament d'aplicacions i utilitza entorns integrats de desenvolupament.
 3. Identifica els diferents tipus de variables i la utilitat específica de cadascun.
 4. Modifica el codi d'un programa per crear i utilitzar variables.
 5. Crea i utilitza constants i literals.
 6. Classifica, reconeix i utilitza en expressions els operadors del llenguatge.
 7. Comprova el funcionament de les conversions de tipus explícites i implícites.
 8. Introduceix comentaris en el codi.
2. Utilitza correctament tipus de dades simples i compostes emprant les estructures de control adients.
 1. Descriu els fonaments de la programació.
 2. Escriu algorismes simples.
 3. Analitza i dissenya els possibles algorismes per resoldre problemes.
 4. Escriu i prova programes senzills reconeixent i aplicant els fonaments de la programació.

-
5. Utilitza estructures de dades simples i compostes.
 6. Escriu i prova codi que faci ús de les estructures de selecció.
 7. Utilitza correctament les diferents estructures de repetició disponibles.
 8. Reconeix les possibilitats de les sentències de salt.
 9. Realitza operacions bàsiques, compostes i de tractament de caràcters.
 10. Revisa i corregeix els errors apareguts en els programes.
 11. Comenta i documenta adequadament els programes realitzats.
 12. Utilitza un entorn integrat de desenvolupament en la creació i compilació de programes simples.

2

Continguts

1. Estructura d'un programa informàtic:

1. Blocs d'un programa informàtic.
2. Projectes de desenvolupament d'aplicacions. Entorns integrats de desenvolupament.
3. Variables. Tipus i utilitat.
4. Utilització de variables.
5. Constants. Tipus i utilitat.
6. Operadors del llenguatge de programació.
7. Conversions de tipus de dades.
8. Comentaris al codi.

2. Tipus de dades simples i compostes. Programació estructurada:

1. Fonaments de programació.
2. Introducció a l'algorísmia.
3. Disseny d'algorismes.
4. Prova de programes.
5. Tipus de dades simples i compostes.
6. Estructures de selecció.
7. Estructures de repetició.
8. Estructures de salt.
9. Tractament de cadenes.

10Depuració d'errors.

11.Documentació dels programes.

12Entorns de desenvolupament de programes.

3

Breu història de Java

Java és un llenguatge de programació de propòsit general **basat en classes**¹ i **orientat a objectes**.

Està dissenyat per ser multiplataforma, és a dir, el codi Java ha de poder executar-se en qualsevol plataforma que suporti Java sens necessitat de recompilar-lo.

Actualment segueix sent un dels llenguatges principals tant per la quantitat de codi escrit com per la quantitat d'ofertes de treball existents.

Des de la seva creació al 1995 Java ha anat evolucionant, a continuació es mostra un resum de les funcionalitats principals de cada versió:

Java 1.0 (1996)

Primera versió pública de Java. Implementava 212 classes organitzades dins vuit paquets. La plataforma de Java sempre ha enfatitzat la retrocompatibilitat, de fet, la majoria de codi escrit per Java 1.0 segueix funcionant en Java 12.

Java 1.1 (1997)

S'afegeixen les "**inner classes**" i la primera versió de la API **Reflection**. La mida de la plataforma es dobla.

Java 1.2 / Java 2 (1998)

S'afegeix la API de **coleccions**, (llistes, mapes, conjunts,...). La gran quantitat de millores de la versió porta a Sun renombrar la plataforma amb el nom de **Java 2**. Es triplica la mida de la plataforma Java.

¹ La herència s'aplica definint classes enlloc de directament sobre els propis objectes

Java 1.3 (2000)

Versió de manteniment basada en la resolució de "bugs" i en la millora de la estabilitat.

Java 1.4 (2002)

Una altra de les versions importants. S'afegeix la API de **i/o**, el suport per **expressions regulars**, **XML** i **XMLST**, per **SSL**, la APil de "**logging**" i el suport per a **criptografia**.

Java 5 (2004)

Apareixen els **genèrics**, els **enums**, les **anotacions**, els mètodes amb arguments variables, **autoboxing** i el bucle **for each**. Aquesta versió mantenia 3562 classes i interfícies en 166 paquets.

Java 6 (2006)

Versió de manteniment basada en la resolució de "bugs" i en la millora de la estabilitat. S'implementa la possibilitat que llenguatges de script interoperin amb Java.

Java 7 (2011)

Primer llançament de Java sota la propietat de **Oracle**. S'introduceix la API **NIO**

Java 8 (2014) (LTS)

Aquesta versió incorpora els canvis de llenguatge més importants des de Java 5. S'introduceixen les **expressions lambda** i s'adeqüen les classes existents, principalment les col·leccions, per l'ús d'aquestes. S'introduceix una nova versió de la API pel tractament de **dates** i **tempo**s i millores importants a les llibreries que gestionen la concorrència.

Java 9 (2017)

S'introduceix el concepte de **modularitat** a la plataforma que permet empaquetar les aplicacions en **unitats de desplegament**. Es modifica l'algoritme de la màquina virtual per defecte i apareix una nova API per treballar amb **processos**.

Java 10 (Març 2018)

És la primera versió en la nova política de llançaments per Java, una nova versió cada sis mesos amb vigència de només sis mesos i una versió LTS cada dos anys.

Java 11 (LTS) (Setembre 2018)

Primera versió LTS des de Java 8. Afegeix poques funcionalitats a Java, alguns canvis interns i una nova API per HTTP.

Java 12 (Març 2019)

Nou recollidor de brossa amb duració constant independentment de la mida de la memòria.

Java 13 (Setembre 2019)

Reimplementa la API `java.net.Socket`.

Java 14 (Març 2020)

Afegeix poques funcionalitats noves.

Java 15 (Setembre 2020)

Afegit suport per literals multi-línia. Nova màquina virtual ZGC de baixa latència i treball concurrent dissenyada per no interrompre els fils d'execució més de 10ms. Aquesta màquina virtual no pretén substituir la GC per defecte

Java 16 (Març 2021)

Millors en la API de Streams. Classes de tipus registre, només contenen dades.

Java 17 (Setembre 2021) (LTS)

Sealed classes.

4

Les parts d'un programa Java

```
public class Hello {  
    public static void main(String[] args) {  
        // Genera una sortida de dades  
        System.out.println("Hello, World!"); // Primera línia  
        System.out.println("How are you?"); // Una altra línia  
    }  
}
```

- `System.out.println("Hello, world!");` és una instrucció que mostra els resultats per la pantalla.
- Java és un llenguatge "**case sensitive**", és a dir, distingeix entre majúscules i minúscules.
- Una funció, o un mètode en el context de la programació orientada a objectes, consisteix en un conjunt d'instruccions a les que se les hi ha posat un nom.
- El programa defineix una funció de nom **main**:

```
public static void main(String[] args) {  
}
```

- Totes les aplicacions Java han de tenir una funció anomenada **main** i definida de la manera anterior.
- La funció `main` determina el punt d'entrada i el punt de sortida del programa Java. Una **classe** consisteix en una col·lecció de mètodes, o funcions en el context de la programació procedimental. Aquest programa defineix una

classe anomenada **Hello**. Per conveni el nom de les classes comença en majúscula.

- Tots els programes Java s'han de posar en fitxers de text amb l'extensió **.java**.
- El nom de la classe ha de coincidir amb el nom del fitxer, per tant, aquesta classe ha d'estar al fitxer **Hello.java**.
- Java utilitza **{ i }** per delimitar un **bloc de codi** que funciona com a una unitat. A l'exemple s'utilitzen els parèntesis per delimitar la classe Hello i per delimitar la funció main.
- Les línies que comencen per **//** són comentaris. Els comentaris són ignorats pel compilador.

Símbols:

//

Indica el principi d'un comentari. El comentari acaba al final de la línia.

{ }

Agrupa un conjunt d'instruccions. Aquesta agrupació permet ser gestionada com a un bloc sencer.

;

Indica el final d'una instrucció. Gairebé totes les instruccions en Java acaben en **;**. Darrera d'un **},** és a dir, d'un bloc, no hi va punt i coma.

""

Delimita un conjunt de caràcters que es tracten sense significat semàntic. Es poden tractar com una dada.

4.1. Compilar el codi font

Compilar és el procés de traduir el codi font en un format binari especial anomenat "bytecode". Això s'aconsegueix amb un programa anomenat compilador, en el cas de java, **javac**, que es distribueix junt amb el JDK d'Oracle.

El procés de compilació de codi java és el següent:



Figura 4.1. Procés de compilació del codi font de Java a bytecode

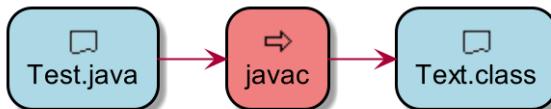


Figura 4.2. Exemple del procés de compilació del codi font de Java a bytecode

Passos necessaris per a poder compilar un programa Java:

1. Cal tenir instal·lat o bé el **Oracle JDK**¹ d'Oracle o bé el **OpenJDK**. És millor instal·lar una de les versions LTS² ja les versions no LTS tenen una vigència de només 6 mesos.
2. Obrir una línia de comandes i situar-se al directori on es troba el fitxer amb el codi font, per exemple.

```
d:\projectes\proves\Test.java
```

3. Per compilar **Test.java** executaríem:

```
d:\projectes\proves\Test.java> javac Test.java
```

4. Si en aquest punt no hem obtingut cap missatge d'error, s'ha compilat correctament el codi font i s'ha generat un nou fitxer anomenat **Test.class** dins al directori proves. Si s'ha obtingut algun error pot ser degut a alguna de les següents raons:
 - a. **Test.java** no està al directori d:\projectes\
 - b. El codi font de **Test.java** té errors de sintaxis.
 - c. El JDK de Java no està instal·lat a la màquina.
 - d. El directori on està instal·lat el JDK no està afegit a la variable d'entorn **PATH** i per tant el sistema **no sap on es troba la comanda javac**.

¹ Java Development Kit

² Long Term Support

Per exemple, si s'ha instal·lat el JDK a **c:\JavaJDK** caldrà afegir la ruta **c:\JavaJDK\bin** a la variable d'entorn PATH.



Si no es vol tocar la variable d'entorn PATH es pot cridar a **javac** posant la ruta sencera d'on es troba.

```
d:\projectes\proves> c:\javaJDK\bin\javac  
Test.java
```

O bé, es podria compilar el programa des del directori **c:\java8** i indicar la ruta sencera a on es troba el codi font.

```
c:> c:\javaJDK\bin\javac d:\projectes\proves  
\Test.java
```

Què és el JDK i què és el JRE

El "**Java Runtime Environment**", el JRE, proporciona les llibreries, la màquina virtual, i altres components per executar aplicacions escriptes en Java.

El "**Java Development Kit**", el JDK, és un superconjunt del JRE i conté tot el que està dins del JRE junt amb les eines necessàries per desenvolupar aplicacions Java, com per exemple els compiladors i els depuradors.

El JDK i l'OpenJDK

L'OpenJDK és la implementació de codi obert de Java SE.

Oracle JDK és mantinguda 100% per la companyia Oracle Corporation, i Open JDK és desenvolupada per la comunitat Java, una part molt petita per Oracle i altres empreses com Red Hat, IBM, Azul systems, Apple Inc o SAP.

Avans de Java 11, en general les dues versions eren iguals amb l'excepció d'algunes llibreries i utilitats que es consideraven funcionalitats comercials.

A partir de Java 11 Oracle ha canviat la llicència d'ús i ha imposat algunes restriccions a la utilització del JDK.

La nova llicència **només permet l'ús gratuït del JDK per ús personal i per desenvolupament.**

Diferents versions de Java, JavaSE, JavaME i JavaEE

JavaSE

És la versió estàndard, utilitzada normalment per software de la banda client i aplicacions "normals".

JavaME

És la versió micro, és bàsicament una versió més petita de Java adequada a processadors amb poca capacitat.

JavaEE

És la versió enterprise o empresarial, utilitzada per programar coses a la banda del servidor.

4.2. Executar el codi compilat

Un programa Java és executat per una JVM³. La JVM s'invoca a través d'un programa anomenat **java.exe**

Per executar el programa **d:\projectes\proves\Test.class** faríem:

```
d:\projectes\proves\java Test
```



Fixeu-vos que per executar un programa Java **no es posa l'extensió .class** del fitxer precompilat.

Què passa quan s'executa la comanda anterior?

1. La JVM intenta localitzar el bytecode de la classe Test
2. La JVM afegeix la paraula ".class" al nom de la classe.
3. Un cop trobat el fitxer físic on es troba el bytecode, la JVM busca la declaració "**public static void main(String[] args)**" dins del fitxer.
 - a. En cas de no trobar la declaració anterior es produiria un error d'execució i es pararia l'execució del programa.
4. Finalment executa el codi que es troba dins de la declaració anterior.

4.3. Tipus d'errors

Quan es programa és molt fàcil produir codi incorrecte que generi algun tipus d'error en la seva execució o que no sigui interpretable pel compilador i no es pugui executar.

Alguns d'aquests errors són detectats pel propi compilador i d'altres no, a més, si l'error és detectat el compilador intenta donar-nos informació per facilitar la feina de detectar-lo, a vegades, però, la informació que en dona pot ser confusa i desconcertant.

En un programa es poden produir tres tipus d'errors:

³ Java Virtual Machine

- Errors en temps de compilació ("compile-time errors")
- Errors d'execució ("runtime errors")
- Errors lògics

4.3.1. Errors en temps de compilació ("compile-time errors")

Aquests errors es produeixen quan no es compleixen les regles de sintaxi del llenguatge Java.

En aquesta situació el programa no pot ser compilat i per tant es genera un missatge d'error en el moment de la compilació i aquesta no es produeix.

Exemples d'aquest tipus d'errors poden ser deixar-se de tancar una clau o un parèntesi o deixar-se un punt i coma al final de la instrucció.

Les missatges d'error del compilador normalment indiquen a on s'ha produït l'error i a vegades també n'indiquen la causa.

Per exemple, el següent codi:

```
1: public class Hello {  
2:  
3:     public static void main(String[] args) {  
4:         System.out.println("Hello, World!")  
5:     }  
6: }
```

Produeix el següent error de compilació:

```
File: Hello.java [line: 4]  
Error: ';' expected
```

Però els missatges d'error no són sempre fàcils d'entendre , a vegades el compilador notifica el lloc on s'ha detectat l'error, no a on s'ha produït. A vegades la descripció de l'error pot ser més confusa que altre cosa. Per exemple:

Per exemple, el següent codi:

```
1: public class Hello {
```

```

2:
3:     public static void main(String[] args {
4:         System.out.println("Hello, World!");
5:     }
6: }
```

Produeix el següent error de compilació:

```

File: Hello.java [line: 6]
Error: reached end of file while parsing
```

4.3.2. Errors en temps d'execució ("runtime errors")

S'anomenen errors en temps d'execució per què no apareixen fins que el programa ha començat a executar-se. A aquest tipus d'errors també se'ls anomena Exepcions, ja que indiquen que alguna cosa excepcional (i dolenta) ha passat.

Per exemple, el intèrpret de Java mostrarà un error si accidentalment s'intenta dividir per zero i mostrarà un missatge semblant al següent:

```

Exception in thread "main" java.lang.ArithmetricException: / by zero at
Hello.main(Hello.java:5)
```

El missatge d'errors dona pistes per la seva correcció, en primer lloc indica el nom de l'excepció *java.lang.ArithmetricException*, a continuació un missatge que indica més específicament que és el què ha passat i finalment es mostra la ubicació i la línia a on s'ha produït l'error.

4.3.3. Errors lògics

Si un programa conté errors lògics compilàrà i s'executarà sense generar missatges d'error, però no funcionarà correctament. Alguna de les instruccions donades pel programador no serà la adequada per obtenir el resultat esperat.

Aquest tipus d'error és el més difícil de detectar ja que normalment no es té l'ajuda ni del compilador ni de l'interpret.

5

Mostrar literals, nombres i caràcters

5.1. Mostrar literals

Considerem el següent fragment de codi:

```
System.out.println("Hello, World!");
```

- Les frases que apareixen entre **cometes dobles** s'anomenen **literals, string literals** en anglès.
- Per mostrar un literal per pantalla s'utilitzen dues comandes:

System.out.println();

Mostra la cadena passada i afegeix un salt de línia al final.

System.out.print();

Mostra la cadena passada sense afegir un salt de línia.

Per exemple:

```
public class Adeu {  
    public static void main(String[] args) {  
        System.out.print("Adéu, ");  
        System.out.println("món cruel");  
        System.out.println("#####");  
    }  
}
```

Mostraria:

```
Adéu món cruel  
#####
```

5.2. Seqüències d'escapament

En una cadena poden aparèixer caràcters que difícilment es poden introduir des del teclat, o bé perquè no tenen una representació gràfica o bé per que trenquen alguna d'elles regles de la construcció de les cadenes.

Per exemple:

- Posar una cometa doble dins d'una cadena.
- Posar un tabulador dins d'una cadena.
- Posar un caràcter que no està representat al teclat però si existeix a la codificació de caràcters utilitzada, normalment UTF-8 en Java, `\t` o `\n` per exemple.

Un caràcter també es pot representar com una seqüència d'escapament.

- Una seqüència d'escapament comença per un `\` i a continuació un caràcter o un codi que fa referència al caràcter que es vol imprimir.

Seqüències de caràcter d'escapament

`\n`

A linefeed

`\r`

A carriage return

`\b`

Backspace

`\t`

Tabulador

`\\"`

Contrabarra

\"

Cometa doble

\'

Cometa simple

També és possible definir un literal de caràcter a partir d'una seqüència d'escapament unicode.

\u0041

A

\u0042

B

etc...

Final de línia

En els sistemes unix, tradicionalment, la finalització d'una línia es denota amb el caràcter \n (new-line).

En Windows, tradicionalment, el salt de línia es denota amb els caràcters \r\n (carriage-return i new-line) per herència dels antics terminals de tipus teletip.

Actualment la majoria de programes accepten el caràcter \n com a final de línia, en particular Java.

Codificació de caràcters

Per aconseguir l'intercanvi d'informació entre dos sistemes cal que un dels sistemes sigui capaç d'entendre de manera no ambigua la codificació dels caràcters representats, en una combinació de bits, i generats per l'altre sistema, i viceversa.

La codificació de caràcters en els sistemes informàtics es basa en els següents fets:

- Cada caràcter es s'associa a un enter no negatiu anomenat "code point".
- Un "character set" és un conjunt de caràcters junt amb els "code points" corresponents.
- El nombre de bits utilitzats per a representar un caràcter determina la quantitat de caràcters diferents que es poden representar en un "charset" determinat

Alguns dels charsets més utilitzats són:

ASCII

- Codifica cada caràcter amb 7 bits, cosa que proporciona 128 caràcters diferents per charset.

ASCII estès

- Codifica cada caràcter amb 8 bits.

UCS

- Utilitza una estructura de 32 bits per a representar un caràcter i pretén proporcionar un únic charset per a tots els llenguatges actuals i passats.
- També proporciona símbols utilitzats en diferents disciplines com ara matemàtiques i física.
- UCS-2, UCS-4, UTF-16, UTF-8 són diferents mètodes de codificació de caràcters dins de UCS.

unicode¹

- Es pot considerar un subconjunt de UCS i codifica cada caràcter amb 16 bits.
- **Els primers 128 caràcters d'unicode són els mateixos que en ASCII**, cosa que facilita la conversió entre els dos sistemes.
- Alguns caràcters poc comuns utilitzen dos "code points" per a ser representats.

5.3. Mostrar caràcters

Java distingeix entre un **literal cadena** i un **caràcter**, en el primer cas es tracta d'un tipus de dades de longitud no definida, en el segon es tracta d'un valor numèric d'exactament dos bytes. Per aquesta raó el tractament intern que es d'un tipus de dades i de l'altre es diferent, la gestió de la memòria no és la mateixa ja que en un cas la longitud és sempre la mateixa i en l'altre no.

Els caràcters es delimiten per **cometes simples** a diferència de les cadenes que es delimiten per cometes dobles.

Les comandes utilitzades per mostrar caràcters són les mateixes que per mostrar literals de tipus cadena:

- `System.out.println('caràcter');`

¹<http://www.unicode.org>

- `System.out.print('caràcter');`

Per exemple:

```
System.out.println('a');
System.out.println('\u0050');
System.out.print('\n');
```



Un caràcter es pot tractar com a caràcter o com a cadena de longitud 1 dependent de si es delimita amb cometes simples o dobles.

5.4. Mostrar nombres

Per mostrar un nombre en Java s'utilitzen les mateixes comandes que per mostrar literals de tipus cadena, en el cas dels nombres però **no es delimiten amb cometes**:

- `System.out.println(nombre);`
- `System.out.print(nombre);`

Els nombres en Java poden ser enters o en coma flotant, també es poden representar en notació científica ².



El separador decimal d'un nombre decimal és el punt, no la coma.



Un nombre es pot tractar com una cadena si es delimita amb cometes dobles o com a un caràcter si té un sol dígit i es delimita entre cometes simples.

No obstant el tractament que es pot fer d'un nombre, d'una cadena o d'un caràcter no és el mateix, per exemple, els nombres es poden operar aritmèticament, les cadenes i els caràcters no.

Per exemple:

² Veure l'apartat dedicat a tipus de dades

```
System.out.println(4);
System.out.println(4.5);
System.out.println(3E5); // 3 x 10^5
System.out.println("49"); // Es tracta el nombre 49 com una cadena
System.out.println(4 * 6); // Correcte 4 x 6 = 24
System.out.println('4' * '6'); // Error, no té sentit multiplicar
    caràcters
```


6

Comentaris i Llegibilitat

Java és un llenguatge de "format lliure", això vol dir que en general es poden posar tants espais i línies en blanc com es vulgui.

No obstant cal tenir en compte que la "composició" d'un programa pot facilitar o dificultar la seva lectura, per exemple, el següent programa és legal però difícil de llegir:

```
public class Lleig{public static void main(String[] args)
{System.out.println("Tela");System.out.println("Més tela");}}
```

Regles per augmentar la llegibilitat del codi:

- Posar cada instrucció en una línia independent.
- Sangrar el programa adequadament. Quan apareix un **{** cal augmentar la sangria de les següents línies, quan apareix un **}** cal reduir la sangria de les següents línies.
- Utilitzar línies en blanc per separar les diferents parts del programa.



Per sagnar les línies és millor utilitzar espais que tabuladors, en general Java aconsella sagnar amb quatre espais.

La majoria d'editors permeten vincular la tecla tabulador a un nombre determinat d'espais.

Seguint aquestes regles, el programa anterior queda:

```
public class Lleig {  
    public static void main(String[] args) {  
        System.out.println("Tela");  
        System.out.println("Més tela");  
    }  
}
```

Encara que els programes de Java siguin llegibles a vegades cal afegir alguna explicació que no forma part del programa pròpiament dit.

En aquest cas Java permet afegir **dos** tipus de comentaris:

Comentaris d'una sola línia

Comencen per // i acaben al final de la línia.

Comentaris multilínia

Comencen per /* i acaben per */ i poden ocupar més d'una línia.

Exemples:

```
// Això és un comentari d'una línia  
// un altre  
  
/* Això és un comentari  
   multi-línia */  
  
System.out.println("Hola"); // Saludem
```

7

Variables i tipus de dades

7.1. Què és una variable

Una variable està formada per un **espai a la memòria principal** de l'ordinador i un **nom simbòlic** o **identificador** que està associat a aquest espai.

En el cas de Java, les variables també estan associades a un **tipus de dades**.

- El nom de la variable es la manera usual de referir-se al valor emmagatzemat.
- El tipus de dades és necessari degut a la naturalesa "**type safe**" del llenguatge Java, és a dir, requereix ser explicit sobre quin tipus d'informació s'està manipulant en cada moment i a canvi es garanteix que la manipulació de les dades es realitza de manera coherent segons la seva naturalesa.

7.2. Què és un identificador

Un identificador en Java és una seqüència de caràcters que s'utilitza per donar nom a alguna construcció del llenguatge.

Els identificadors en Java segueixen les següents regles:

- No poden començar amb un dígit.
- Tots els caràcters han de ser lletres, números, guió baix _ o signe de dolar \$ (El símbol \$ està reservat per casos especials i no s'hauria d'utilitzar)
- Teòricament poden tenir longitud arbitrària.
- Són "case sensitive".

- Tot i que en principi són identificadors vàlids evitarem utilitzar caràcters dependents de l'idioma, com ara lletres amb accent, la c trencada, etc...

```
// Identificadors vàlids
num1
kn
_abc
resultatOperacio
resultat_Operacio

// Identificadors invàlids
2num
my name
num1+num2
num1-num2
```

Java defineix un a llista de paraules reservades que tenen significats predefinits i que només es poden utilitzar en contextos definits pel propi llenguatge. Aquestes paraules clau no es poden utilitzar com a identificadors.

Taula 7.1. Llista de paraules reservades en Java

abstract	continue	for	new	switch
assert	goto	package	synchronized	boolean
do	if	private	this	break
double	implements	protected	throw	byte
else	import	public	throws	case
enum	instanceof	return	transient	catch
extends	int	short	try	char
final	interface	static	void	class
finally	long	strictfp	volatile	const

Les paraules **const** i **goto** estan reservades però no s'utilitzen actualment en Java.

Les paraules **true**, **false** i **null** també estan reservades i no es poden fer servir com a identificadors.

7.3. Què és un tipus de dades

Un tipus de dada queda definit per:

- Un conjunt de valors
- Un conjunt d'**operacions** que es poden aplicar a tots els valors del conjunt
- Una representació de les dades, que determina com s'emmagatzemen els valors.
- Un llenguatge de programació proporciona un conjunt de **tipus predefinits**.
- Un llenguatge de programació pot permetre als programadors definir nous tipus de dades.
- Java suporta dues categories de tipus de dades:
 - Tipus de dades **primitius**.
 - Tipus de dades **per referència**

7.4. Diferències entre tipus primitius i tipus per referència



Formalment, el paradigma d'orientació a objectes sobre el que es fonamenta el llenguatge Java implica que totes les dades gestionades pel llenguatge siguin de tipus per referència, en el marc teòric de la orientació a objectes els tipus primitius no tenen massa sentit.

La decisió de Java de mantenir dos tipus de dades diferents està motivada per temes d'eficiència, els tipus primitius són molt més ràpids de "gestionar" que els tipus per referència.

Malaüradament significa que cal aprendre dos conjunts de regles sobre el funcionament de les dades en Java

La diferència entre un tipus primitiu i un tipus per referència és:



- Una variable de tipus primitiu conté un valor.
- Una variable de tipus per referència conté l'adreça de memòria a on s'ubica la dada que conté.

7.5. La gestió de la memòria en Java (el "stack" vs el "heap")

La majoria de llenguatges de programació, Java entre ells, defineix dues àrees de memòria diferenciades, la pila o "stack" i el pilot ¹ o "heap" el tractament intern que es fa d'aquestes dues àrees és diferent.

7.5.1. El "stack"

La **pila** és una regió especial de la memòria de l'ordinador que emmagatzema variables temporals creades per a cada funció, en particular per la funció **main**.

Té una estructura de tipus **LIFO**, "Last Input First Output", que està optimitzada per la CPU.

Quan una funció declara una nova variable aquesta és afegida "al damunt" de l'estructura LIFO. Cada cop que una funció finalitza **totes** les variables declarades en aquella funció s'eliminen i s'allibera l'espai de memòria que ocupaven. L'àrea de memòria alliberada queda disponible per a noves variables declarades a noves funcions.

L'avantatge principal d'utilitzar la pila per emmagatzemar variables és que la gestió de la memòria és automàtica i el fet de treballar en una estructura de pila fa que la creació i alliberament de la memòria sigui molt ràpida.

Per altra banda:

- La quantitat de memòria dedicada a la pila està limitada.
- El tractament LIFO que fa la pila impedeix que un element situat al mig de la pila pugui créixer, solaparia la memòria del següent element, per tant, aquesta estructura només pot emmagatzemar elements que no canvien de mida durant la seva existència.

7.5.2. El "heap"

El "heap" és una regió de memòria no gestionada automàticament, que a diferència de el "stack" no té restriccions en la mida de les variables que pot contenir, llevat de la pròpia mida de la RAM de l'equip.

¹ És una traducció lliure

L'estructura del "heap" és "lliure", de manera que es pot emmagatzemar un valor a qualsevol direcció de memòria de l'estructura, això fa que:

- Abans d'emmagatzemar un valor al "heap" caldrà buscar i trobar un espai de memòria de suficient capacitat. Cosa que no feia falta al "stack" i per tant repercutix negativament al rendiment.
- Els elements emmagatzemats al "heap" no s'eliminen automàticament al sortir de la funció que els utilitza, en alguns llenguatges s'han d'alliberar manualment, C, C++, i en d'altres un element específic de la màquina virtual, anomenat **recol·lector de brossa** serà l'encarregat de fer-ho quan detecti que no s'utilitzen més.
- Com que la reserva i l'alliberament de la memòria és manual, la memòria tendeix a quedar fragmentada, cosa que té un impacte negatiu al rendiment.

7.5.3. Quina de les estructures utilitzar en Java?

Java no permet al programador triar directament quina de les estructures de memòria utilitzar.

El **tipus de dades** que es vol emmagatzemar a la memòria determina quina de les estructures s'utilitza.

- Els **tipus de dades primitius** s'emmagatzemen al "stack".
- Els **tipus de dades per referència** s'emmagatzemen al "heap" i es manté la direcció de memòria on s'ubiquen al "stack".

7.5.4. En resum

Les característiques de cada estructura són:

Stack

- Accés molt ràpid.
- La memòria no es fragmenta.
- La reserva i l'alliberament de la memòria són automàtics.
- Les variables tenen un temps de vida definit per la funció a on es declaren.
- L'estructura té una mida limitada.
- Les variables no poden canviar de mida.

Heap

- Les variables són accessibles globalment.
- No hi ha límit en la mida de la memòria.
- L'accés és relativament lent.
- No es garanteix un ús eficient de l'espai, la memòria es fragmenta a mesura que s'emmagatzemen i s'alliberen blocs de memòria. La gestió de la memòria és manual. En Java ho tractarà la pròpia màquina virtual amb l'ajuda d'un recol·lector de brossa automatitzat.

7.6. Declaració de variables

Abans d'emmagatzemar un valor en una variable, en Java, cal indicar el tipus de dades que emmagatzemarà i el nom pel que li farem referència, **declarar una variable** consisteix precisament en fer les dues accions anteriors.

La sintaxi és:

tipus nom;

Per exemple.

```
int i;
String nomClient;
int hores, minuts;
```



Alguns tipus comencen amb majúscula i d'altres no.



Un cop s'ha declarat una variable amb un nom **no** es pot tornar a declarar una altra variable amb el mateix nom dins del mateix àmbit, té sentit, ja que si es pogués tindriem un mateix nom que faria referència a dues adreces de memòria diferents.

Per exemple,

```
int valor;
valor = 3;
```

```
// Qualsevol de les següents línies produeix un
// error
char valor = 'a';
int valor;
int valor = 10;
```

7.7. Assignació de variables

Un cop s'ha declarat una variable cal donar-li un valor. Aquest pas s'anomena **assignació de variables**.



Una variable **declarada**, **no** es pot fer servir fins que no se li assigni algún valor. El següent codi donarà un error:

```
public class Exemple {
    public static void main(String[] args) {
        int a;
        System.out.println(a);
    }
}

java.lang.RuntimeException: variable a has not been
assigned a value
```



L'assignació de variables depèn del tipus de dades, **els tipus primitius i els tipus per referència s'assiguen de forma diferent**.

7.8. Assignació de variables pels tipus primitius

Per assignar un valor a una variable d'un tipus primitiu s'utilitza l'operador **=** seguit del valor a assignar, per exemple:

```
int i; // Declaració de la variable i de tipus int
i = 10; // Assignació del valor 10 a la variable i
```

També és possible fer els dos passos anteriors en un de sol:

```
int i = 10; // Declaració i assignació de la variable i de tipus int
```



Les variables es poden assignar tantes vegades com es consideri necessari, en aquest cas *una nova assignació d'una variable sobreescriva l'últim valor que aquesta pogués tenir.

Per exemple,

```
int valor;
valor = 10;
valor = 23;
valor = 5;

// El valor de la variable valor en aquest punt és 5
```

7.9. Assignació de variables pels tipus per referència

Per assignar un valor a una variable de tipus per referència cal:

- Crear una nova instància del tipus mitjançant l'operador **new**
- Assignar la nova instància creada a la variable.

```
String s; // Declaració de la variable i de tipus int
s = new String(); // Assignació del valor 10 a la variable i
```



El cas del tipus de dades **String** és especial, tot i ser un tipus per referència permet la declaració i l'assignació de valors com si es tractés d'un tipus per valor.

Es a dir,

```
String s1 = "és una cadena"; // És correcte tot i
    ser un tipus per referència
String s2 = new ("és una cadena"); // És correcte
    per ser un tipus per referència
```

7.10. Tipus primitius en Java

Taula 7.2. Tipus primitius en Java

Nom	Valor	Mida	Rang
-----	-------	------	------

byte	Enter	1 byte	-128 fins a 127
short	Enter	2 bytes	-32,768 fins a 32,767
int	Enter	4 bytes	-2^{31} fins a $2^{31} - 1$
long	Enter	8 bytes	-2^{63} fins a $2^{63} - 1$
float	Coma flotant	4 bytes	$\pm 3.40282347 \times 10^{+38}$ to $\pm 1.40239846 \times 10^{-45}$
double	Coma flotant	8 bytes	$\pm 1.79769313486231570 \times 10^{+308}$ fins a $\pm 4.94065645841246544 \times 10^{-324}$
char	Un caràcter (Unicode)	2 bytes	Valors Unicode de 0 a 65,535 (\u0000 a \xFFFF)
boolean		1 bit	true o false

7.10.1. Tipus Enters

- Un tipus de dades enter pot contenir valors numèrics enters.
- Java proporciona cinc tipus enters: **byte**, **short**, **int**, **long**, i **char**.

7.10.2. Tipus int

- Ocupa 32 bits.
- És un tipus amb signe.
- El rang vàlid és -2^{31} a $2^{31} - 1$.
- Els literals enters es poden representar en format en funció d'un prefix:
 - Decimal
 - Octal (**prefix 0**)
 - Hexadecimal (**prefix 0x**)
 - Binari (**prefix 0b**)

Representació d'enters en diferents bases de numeració.

```
int num1 = 17;           // Base 10
int num2 = 021;          // Octal
```

```
int num3 = 0x123;      // Base 16
int num4 = 0xdecafe; // Base 16
int num5 = 0b10101;   // Base 2
int num6 = 0b00011;   // Base 2
```

La representació binaria dels nombres enters

Els ordinadors utilitzen el sistema de numeració binari per emmagatzemar la informació. Tota la informació s'emmagatzema utilitzant els símbols 1 i 0, anomenats **bits** (binary digits). ²

Un nombre enter positiu s'emmagatzema en Java en la seva representació binària.

Per exemple, el número 13 en una variable de tipus byte serà **00001101₂**.

La complicació es troba en com cal emmagatzemar els nombres negatius, per optimitzar els càlculs matemàtics els nombres enters s'emmagatzemen codificats en una codificació anomenada **complement a 2**.



El complement a 2 d'un nombre binari representat en N bits és un nombre binari representat en N bits tal que sumat al primer dona 2^N .

Com que 2^N és de l'estil 10...0 amb N zeros, si es consideren només els N últims dígits de la suma resulta que la suma del nombre original més el seu complement a 2 és 0.

² Veure apèndix 1

Per exemple,

El complement a 2 de **00001101** serà **11110011** ja que:

$$\begin{array}{r} 00001101 \\ +11110011 \\ \hline 100000000 \end{array}$$

Si considerem que els nombres són de tipus byte i el resultat de la suma anterior és **00000000**.



El complement a 2 d'un nombre binari **B** es pot calcular directament invertint els zeros i els uns de **B** i sumant 1.

Per exemple,

$$52_{10} = 00110100_2$$

$$\begin{array}{r} 11001011 \\ +00000001 \\ \hline 11001100 \end{array}$$

7.10.3. Tipus long

- Ocupa 64 bits.
- És un tipus amb signe.
- El seu rang és de -2^{63} a $2^{63} - 1$.
- En cas d'ambigüitat entre un literal de tipus **int** i un literal de tipus **long** es pot afegir una **L** al literal per indicar que és del segon tipus.

Representació de nombres de tipus long.

```
long num1 = 0L; // Tipus long
long num2 = 401L; // Tipus long
int num3 = 401; // Tipus enter
```

```
long num1 = 031L; // Format octal
long num1 = 0X19L; // Format hexadecimal
long num1 = 0b11001L; // Format binari
```

7.10.4. Tipus byte

- Ocupa 8 bits.
- És un tipus amb signe.
- El seu rang és de -128 a 127.

7.10.5. Tipus short

- Ocupa 16 bits.
- És un tipus amb signe.
- El seu rang és de 2^{15} a $2^{15} - 1$.

7.10.6. Tipus char

- Ocupa 16 bits.
- És un tipus **sense signe**.
- El seu rang és de 0 a 65535 ($= 2^{16} - 1$).
 - El seu rang de valors correspon justament al del conjunt de caràcters unicode.

Un caràcter s'expressa posant-lo entre **cometes simples**.

El següent exemple assigna valors a variables de tipus char:

```
char c1 = 'A';
char c2 = 'L';
char c3 = '5';
char c4 = '/';
```



Una seqüència d'un o més caràcters delimitats per cometes dobles és un valor de tipus **String** i per tant la següent sentència donarà error:

```
char a = "A"; // Error!!, "A" és de tipus String i
a és de tipus char
```

7.10.7. Tipus boolean

- El tipus boolean només admet dos valors: **true** i **false**. Aquests dos valors s'anomenen valors booleans.

```
boolean procesFinalitzat;
procesFinalitzat = true;
```

7.10.8. Tipus en coma flotant

- Un nombre que conté una part fraccional es coneix com a nombre real, per exemple 0.4, 3423.534
- Un ordinador emmagatzema els nombres en format binari, siguin enters o reals.
 - Emmagatzemar un nombre enter és relativament fàcil, essencialment és fer un canvi de base.³
 - Emmagatzemar un nombre real implica representar la coma decimal dins de la representació binària del nombre cosa que no és un procés trivial.

Existeixen dues estratègies per emmagatzemar un nombre real a la memòria de l'ordinador:

Representació en coma fixa

Aquesta estratègia assumeix que sempre hi ha un nombre fix de dígits abans i després de la coma decimal.

Representació en coma flotant

Aquesta estratègia permet emmagatzemar la posició de la coma decimal en el nombre binari.

³ Això és cert pels enters positius, els enters negatius han de tenir en compte el signe i la conversió a binari no és tan directe

Aquesta representació és menys acurada i més lenta que la representació en coma fixa, no obstant permet emmagatzemar un rang més elevat de nombres.

Java proporciona dos tipus de dades de coma flotant: **float** i **double**.



És important notar que no tots els nombres reals admeten una representació binària exacta i per tant es codifiquen com a nombres en coma flotant de manera aproximada. **Això pot produir errors en la exactitud dels càlculs que utilitzen aquests tipus de dades.**

Nombres en coma flotant

Un nombre en coma flotant està format per quatre parts:

1. Signe
2. Mantissa
3. Base
4. Exponent

Per exemple, el nombre **19.25** es pot representar com a nombre en coma flotant com **+ 19.25 x 10⁰**, a on:

1. Signe = positiu
2. Mantissa = 19.25
3. Base = 10
4. Exponent = 0

No obstant el nombre anterior es pot representar d'altres maneres a la mostrada anteriorment:

```
19.25 x 100
1.925 x 101
0.1925 x 102
192.5 x 10-1
1925 x 10-2
```

En Java els nombres en coma flotant es representen com els exemples anteriors però substituint la **base** per la lletra **E** i eliminant el símbol de la multiplicació.

Els següents nombres són correctes en Java i tots representen al nombre **19.25**.

```
19.25E0
1.925E1
1.925E+1
0.1925E2
192.5E-1
1925E-2
```

7.10.9. Tipus float

- Utilitza 32 bits per emmagatzemar un nombre en coma flotant seguint l'estàndard IEEE 754.
- El seu rang oscil·la entre $\pm 1.4 \times 10^{-45}$ i $\pm 3.4 \times 10^{38}$ aproximadament.
- Un literal numèric acabat en **f** o en **F** es considera de tipus float.
- A la hora d'assignar un valor a una variable de tipus float es pot utilitzar notació científica.

Exemples:

```
// Són tots correctes
float f1 = 8F;
float f2 = 8.0F;
float f3 = 3.51F;
float f4 = 0.0F;
float f5 = 16.78f;
float f6 = 32.5E-1F;
float f7 = 0.325E+1F;
float f8 = 0.0325E2F;
```



El tipus de dades float defineix una sèrie de valors útils:

```
float f1 = Float.POSITIVE_INFINITY;
float f2 = Float.NEGATIVE_INFINITY;
float f3 = Float.NaN;
float f4 = Float.MAX_VALUE;
float f5 = Float.MIN_VALUE;
```

7.10.10. Tipus double

- Utilitza 64 bits per emmagatzemar un nombre en coma flotant seguint l'estàndard IEEE 754.
- El seu rang oscil·la entre $\pm 4.9 \times 10^{-324}$ i $\pm 1.7 \times 10^{308}$ aproximadament.
- Un literal numèric acabat en **d** o en **D** es considera de tipus double.
- A la hora d'assignar un valor a una variable de tipus double es pot utilitzar notació científica.

```
// Són tots correctes
double d1 = 8D
double d2 = 8. ;
double d3 = 8.0;
double d4 = 8.D;
double d5 = 78.9867;
double d6 = 45.0;
double d7 = 32.5E-1;
double d8 = 0.325E+1;
double d9 = 0.325E1;
double d10 = 0.0325E2;
double d11 = 0.0325e2;
double d12 = 32.5E-1D;
double d13 = 0.325E+1d;
double d14 = 0.325E1d;
double d15 = 0.0325E2d;
```



El tipus de dades float defineix una sèrie de valors útils:

```
double f1 = Float.POSITIVE_INFINITY;
double f2 = Float.NEGATIVE_INFINITY;
double f3 = Float.NaN;
double f4 = Float.MAX_VALUE;
double f5 = Float.MIN_VALUE;
```



Un nombre en coma flotant al que no s'ha indicat el tipus de dades, float o double, és per defecte double.

Això produeix que algunes expressions puguin donar errors inesperats. Per exemple:

```
float f = 1.0/2; // Error, ja que considera que 1.0
és un double i per tant el resultat de l'operació
també és un double.
float f = 1.0f/2 // Correcte
```

8

Operadors

Un operador és un símbol que realitza una determinada operació sobre un, dos o tres operands, i produeix un resultat. O al revés, **produeix un resultat** i realitza una determinada operació.

Els operadors es poden classificar segons dos criteris:

- El nombre d'operands sobre els que operen.
- El tipus d'operació que realitza sobre els operands.

Hi ha tres tipus d'operadors segons el nombre d'operands. Aquests operadors s'anomenen **unari**, **binari** o **ternari** en funció del nombre d'operands.

Segons el tipus d'operació que realitzen es poden classificar en:

- Operador d'assignació
- Operadors aritmètics
- Operador de concatenació de cadenes
- Operador de "casting"
- Operadors relacionals
- Operadors lògics o booleans
- Operador ternari
- Operadors a nivell de bit

8.1. Operador d'assignació

L'operador d'assignació és un operador binari que s'utilitza per assignar un valor a una variable.

- El valor de la dreta es assignat a la variable de l'esquerra.
- L'operand de l'esquerra ha de ser una variable, el de la dreta pot ser un valor o una variable.
- En Java l'operador d'assignació és denota per `=`.



L'operador `=` no correspon a la igualtat matemàtica, si fos així, l'expressió `a=b` voldria dir que `a` i `b` són iguals per sempre més, per tant **canviar el valor de `b` canviaria també el valor de `a` i viceversa**.

Per exemple:

```
int num;
num = 25;
System.out.println(num); // Mostra el valor 25
```

Java s'assegura que el valor de la dreta és compatible amb el tipus de dades de l'esquerra, en cas contrari es produeix un error de compilació.



Què vol dir que el valor de la dreta és compatible amb el valor de l'esquerra?

De forma general vol dir que el tipus de dades del valor de la dreta és el mateix o **es pot incloure** dins de la variable de l'esquerra.

Per exemple:

```
byte b = 5;
char c = 'a';
short s = -200;
int i = 10;
i = b; // Ok. Un byte sempre es pot posar dins d'un int
i = c; // Ok. Un char sempre es pot posar dins d'un int
i = s; // Ok. Un short sempre es pot posar dins d'un int
```

```
long big = 524L;
float f = 1.19F;
int i = 15;
i = big; // Error de compilació. Un long no té perquè caber dins d'un
        int
i = f; // Error de compilació. Un float és un número en coma flotant i
        un int no.
```

El valor de la dreta pot ser una variable:

```
int num1 = 100;
int num2 = 200;
num1 = num2; // num1 és 200. num2 és 200
num2 = 500; // num2 és 500. num1 segueix essent 200
```

8.2. Operadors aritmètics

S'utilitzen sobre els tipus de dades byte, short, char, int, long, float, i double.

Taula 8.1. Llista dels operadors aritmètics en Java

Operadors	Descripció	Tipus	Exemple	Resultat
+	suma	Binari	2 + 5	7
-	resta	Binari	5 - 2	3
+	més	Unari	+5	+5
-	menys	Unari	-5	-5
*	multiplicació	Binari	2 * 5	10
/	Divisió	Binari	5 / 2	2, COMPTE!! depèn dels tipus de dades
%	mòdul	Binari	5 % 3	2, COMPTE!! Dona resultats poc intuïtius amb nombres negatius

++	increment	Unari	num++	Avalua num i incrementa el valor
++	increment	Unari	++num	Incrementa el valor i avalua num
--	decrement	Unari	num--	Avalua num i decrementa el valor
--	decrement	Unari	--num	Decrementa el valor i avalua num
+=		Binari	num += 2	Suma 2 a num i assigna el resultat a num
-=		Binari	num -= 5	Resta 5 a num i assigna el resultat a num
*=		Binari	num *= 10	Multiplica num per 10 i assigna el resultat a num
/=		Binari	num /= 3	Divideix num entre 3 i assigna el resultat a num
%=		Binari	num %= 2	Calcula num % 2 i assigna el resultat a num.

8.3. Particularitats de l'operador /

L'operador / funciona de dues maneres diferents en funció del tipus de dades dels operands.

- Si els dos operands de la divisió són enters, això és, byte, short, char, int, o long la divisió és una **divisió entera** i el resultat és un enter.
- Si un dels dos operands és de coma flotant la **divisió és decimal** i el resultat és un número en coma flotant.

Per exemple:

```
num = 5/2; // Dona 2
num = 5.0/2.0 // Dona 2.5
num = 5f/2f // Dona 2.5
```

8.4. Errors d'arrodoniment

La majoria de nombres en coma flotant són "aproximadament correctes". Alguns nombres no es poden representar exactament com a un nombre en coma flotant, per exemple els nombres periòdics.

- Cal anar en compte amb els errors d'arrodoniment quan s'opera amb aquests tipus de nombres.
- Cal anar en compte quan s'intenta determinar si un nombre en coma flotant és zero.

Per exemple:

```
System.out.println(0.1 * 10); // Dona 1.0
System.out.println(0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1);
```

```
0.9999999999999999
```

Un altre exemple:

```
double valor = 0;
```

```
valor += 5.6;
valor += 5.8;

System.out.println(valor);
```

11.399999999999999

i un altre:

```
System.out.println(0.1d + 0.2d);
```

0.30000000000000004

8.5. Jerarquia dels operadors

Considereu el següent fragment de codi.

```
int resultat = 10 + 8 / 2; // Quin és el valor que s'assignarà a
resultat?
```

El resultat serà 9 o 14 en funció de quina és la operació que es computa en primer lloc.

En aquest cas el resultat serà 14 ja que l'operador / té preferència respecte l'operador - i per tant es computa en primer lloc.

Considereu una altra expressió:

```
double resultat = 10 * 5 / 2;
```

En aquest cas els dos operadors tenen la mateixa precedència i per tant l'operació es calcula d'esquerra a dreta. El resultat és 25.



Els diferents operadors tenen diferents preferències que determinen quins es computen en primer lloc.

En cas d'operadors amb la mateixa preferència es computen d'esquerra a dreta.

Es pot modificar la precedència dels operadors utilitzant **parèntesis**, una operació entre parèntesis s'avalua sempre en primer lloc.

- En cas de tenir parèntesis "niuats" es computen en primer lloc els que estan més a dins.

Exemples:

```
2 + 5 * 4 = 22
(2 + 5) * 4 = 28
2 * 8 / 2 + 3 + 3 = 14
2 * 8 / (2 + 3 + 3) = 2
2 * ((3 + 1) * 2 + 1) = 18
2 * (3 + 1 * 2 + 1) = 12
2 * 3 + 1 * 2 + 1 = 9
-3 + 4 = 1
-(3 + 4) = -7
```

Donat l'elevat nombre d'operadors en Java la seva jerarquia és llarga i complicada. En cas de dubte:

- Feu la prova, o bé,
- Poseu parèntesis, o bé,
- Desglosseu la operació en operacions més petites.

8.6. Operador de concatenació de cadenes

L'operador **+** està **sobrecarregat**. Diem que un operador està **sobrecarregat** si s'utilitza per a realitzar més d'una funció.

L'operador **+** es pot utilitzar sobre tipus de dades **String** o **literals de cadena**, en aquest cas, s'interpreta com a **concatenació**.

Exemples:

```
System.out.println("Hola " + "Joan"); // Hola Joan
```

```
String s1 = "Hola";
string s2 = "Joan";
String s3 = s1 + s2; // s3 = "HolaJoan"
System.out.println(s3); // HolaJoan
```



Encara més, l'operador **+** està redefinit per operar sobre un **operand String** i un **operand de qualsevol tipus primitiu**.

En aquest cas es transforma el tipus primitiu en una representació de tipus cadena i es concatena amb l'operand cadena.

El resultat de la operació és un String.

Exemples:

```
System.out.println(23 + " anys"); // 23 anys

String s = 0f + " graus" // s = "0.0 graus"

String descripcioArticle = "Llibre";
int preuArticle = 10;
System.out.println("Article: " + descripcioArticle + ", Preu: " +
    preuArticle + " eur") // Article: Llibre, Preu: 10 eur
```

8.7. Conversions de tipus entre dades numèriques (Castings)

Assignar un valor d'un tipus a una variable d'un tipus diferent es coneix amb el nom de **casting de tipus**

Exemple:

```
int i = 10;
long l = i; // Convertim un enter en un long, java ho deixa fer
automàticament.
System.out.println(l); // 10
float f = i; // Convertim un int en un float, Java ho deixa fer
automàticament.
System.out.println(f); // 10.0
```

```
byte b = i; // Error, no es possible convertir automàticament un enter  
en un byte.
```

En funció dels tipus de dades involucrats a la conversió s'en poden distingir dos tipus:

- Conversió implícita
- Conversió explícita

8.8. Conversió implícita de tipus

La conversió implícita de tipus es dona quan:

- els dos tipus **són compatibles**.
- el tipus objectiu és més gran (en bits) que el tipus inicial.

En aquest cas Java permet la conversió de forma automàtica.



La conversió implícita entre tipus numèrics segueix la següent jerarquia:

byte → short → int → long → float → double

8.9. Conversió explícita de tipus

Java no permet assignar un tipus més gran (en bits) a un tipus compatible més petit ja que podria haver-hi una pèrdua d'informació, el valor del tipus gran pot no poder-se emmagatzemar dins del tipus petit.

En aquest cas és el programador qui pren la responsabilitat de fer el canvi de tipus de manera explícita.

Per fer-ho afegeix, entre parèntesi, el tipus de dades objectiu just després de l'operador d'assignació.

Exemples:

```
int i = 100;  
long l = i; // Error
```

```
int i = 100;
long l = (long) i; // OK, casting explicit
System.out.println(l); // 100
```

```
int i = 65536;
byte b = i; // Error
byte b = (byte) i; // OK però perdem dades!!
System.out.println(b) // 0
```

9

Constants

Les **constants** o **literals** representen noms per a valors específics.

A diferència de les variables les constants no canvien el valor.

9.1. Constants amb nom

Una constant amb nom és un identificador que representa un valor de forma permanent.

Per conveni el nom de les constants s'escriu en majúscules.

Treballar amb constants aporta una sèrie de avantatges:

- No cal escriure el mateix valor si s'utilitza múltiples vegades.
- Si es canvia el valor de la constant només s'ha de fer en un únic punt del codi font.
- El nom descriptiu de la constant facilita la comprensió del codi font.

La paraula clau **final** s'utilitza per indicar que una declaració és invariant, és a dir, no es podrà canviar durant l'execució del programa.

```
public class CalcularAreaCercle {  
    public static void main(String[] args) {  
        final double PI = 3.1416;  
        double radi = 20;  
  
        double area = radius * radius * PI ;  
    }  
}
```

```
    System.out.println("L'ares del cercle de radi " +radius + " és " +
area);
}
}
```

10

System.in, System.out i System.err

Tots els sistemes operatius gestionen tres fluxos de dades comunament anomenats **standard input**, **standard output**, i **standard error** que representen canals de comunicació entre el programa en execució i l'entorn d'execució.

Si el programa s'executa en un terminal de text els fluxos anteriors estan enllaçats al terminal en el cas de **standard input** i **standard error** i al teclat en el cas de **standard output**.

A més, internament els tres fluxos estan numerats de la següent manera:

- standard input → 0
- standard output → 1
- standard error → 2

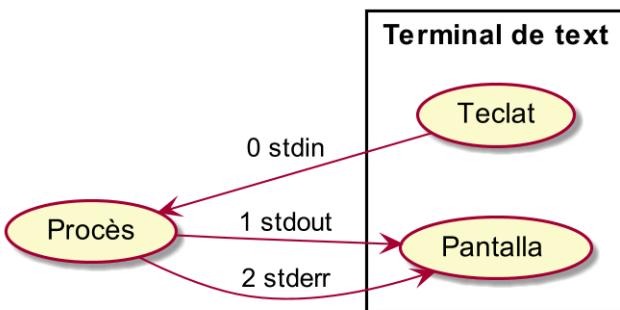


Figura 10.1. Fluxos estàndard d'entrada i sortida

Java permet accedir als tres fluxos de dades anteriors a través de:

-
- System.out
 - System.in
 - System.err

De manera que per escriure bytes al "standard output" faríem

```
public class Streams {  
    public static void main(String[] args) {  
        byte value;  
        value = 65;  
        System.out.print(value);  
    }  
}
```

Per escriure bytes al "standard error" faríem

```
public class Streams {  
    public static void main(String[] args) {  
        byte value;  
        value = 65;  
        System.err.print(value);  
    }  
}
```

I per llegir un byte desde el "standard input" faríem

```
public class Streams {  
    public static void main(String[] args) throws java.io.IOException  
    { ①  
        int value = System.in.read(); ②  
        System.out.println(value);  
    }  
}
```

- ➊ De moment no ens preocupa la sinàxi d'aquesta línia.
- ➋ Java ens obliga a llegir enters enlloc de bytes, és un tema tècnic, podem considerar que el que llegim són bytes.

El problema principal amb els fluxos de dades natius, sobretot amb el d'entrada, és que treballen directament en bytes cosa **que fa difícil el tractament de la informació** ja que caldrà anar convertint les seqüències de bytes al tipus

de dades esperat. **Per aquesta raó no utilitzarem System.in directament si volem obtenir valors d'entrada des de la terminal.**

Redirecció dels fluxos estàndard

Els fluxos estàndard es poden redireccionar, una manera de fer-ho és operant amb el programa directament des del terminal. Per exemple:

Per a redireccionar **stdout** s'utilitza el símbol **>**, de fet **>** és una abreviatura de **1>**.

```
c:\> echo hola  
hola  
c:\> echo hola 1> fitxer.txt  
c:\> type fitxer.txt  
hola
```

Per a redireccionar **stderr** s'utilitza el símbol **2>**.

```
c:\> echo hola  
hola  
c:\> echo hola 2> fitxer.txt  
hola ①  
c:\> type fitxer.txt
```

- ① En aquest cas hola s'escriu a la sortida estàndard i per tant el fitxer estarà buit.

Per a redireccionar **stdin** s'utilitza el símbol <.

Prenent el següent programa:

```
public class Streams {  
    public static void main(String[] args) throws  
        java.io.IOException {  
        int value = System.in.read();  
        System.out.println(value);  
    }  
}
```

```
c:\> echo A > fitxer.txt  
c:\> type fitxer.txt  
A  
c:\> java Streams < fitxer.txt  
65
```

11

Entrada i sortida de dades per la Consola

11.1. Sortida per pantalla amb format

Quan mostrem un double per pantalla es mostren fins a 16 decimals.

```
System.out.print(4.0 / 3.0);
```

El resultat mostra:

```
1.3333333333333333
```

System.out proporciona un altre mètode anomenat **printf** que dona més control sobre el format de sortida.

Per exemple:

```
System.out.printf("Quatre terços = %.3f", 4.0 / 3.0);
Quatre terços = 1,333
```

%.3f és una cadena de format, especifica de quina manera de mostrar la sortida.

Aprendre com funcionen les cadenes de format de **printf** és com aprendre un llenguatge dins de Java, hi ha moltes opcions i els detalls són complicats.

A continuació es mostra un resum de les possibilitats:

Cadenes de format

Tenen la següent estructura:

% [flags] [ample] [.precisió] caràcters-de-conversió

Flags

- alineat a l'esquerra (per defecte s'alinea a la dreta)

+

Mostra + o - en funció del signe del valor numèric.

0

Mostra els valors numèrics amb zeros al davant (per defecte es mostren blancs)

,

Separador de milers etc...

..

Un espai mostra el signe - si el nombre és negatiu o un blanc en cas contrari.

Ample

Representa la quantitat mínima de caràcters que es mostren a la sortida.

Precisió

Especifica el nombre de díigits de precisió en els valors de coma flotant (amb arrodoniment) o bé la longitud d'una subcadena extreta d'una cadena.

caràcters-de-conversió

d

Enter [byte, short, int, long]

f

Nombre en coma flotant [float, double]

c

char. C indica majúscules.

s

String. S indica majúscules.

h

hashcode. Útil per imprimir referències.

Els salts de línia són dependents de plataforma. Utilitzar `%n` en lloc de `\n` per millor compatibilitat.

Exemples:

```
public static void main(String[] args) {
    long n = 461012;
    System.out.printf("%d%n", n);           // --> "461012"
    System.out.printf("%08d%n", n);         // --> "00461012"
    System.out.printf("%+8d%n", n);          // --> "+461012"
    System.out.printf("%,8d%n", n);          // --> " 461,012"
    System.out.printf("%+,8d%n%n", n);       // --> "+461,012"

    double pi = Math.PI;

    System.out.printf("%f%n", pi);           // --> "3.141593"
    System.out.printf("%.3f%n", pi);         // --> "3.142"
    System.out.printf("%10.3f%n", pi);        // --> "      3.142"
    System.out.printf("%-10.3f%n", pi);       // --> "3.142"
    System.out.printf(Locale.FRANCE,
                      "%-10.4f%n%n", pi); // --> "3,1416"
}
```

11.2. Entrada de dades, classe Scanner

La classe **Scanner** es pot utilitzar per obtenir dades tant des de fitxers com des del teclat.

Només utilitzarem la classe Scanner per obtenir dades des de teclat.



La classe Scanner es troba dins d'un paquet anomenat **java.util** per utilitzar aquesta classe caldrà indicar a l'entorn Java on es troba. Per fer-ho utilitzarem la comanda **import** de la següent manera:

```
import java.util.Scanner;
```

Els "import" sempre es troben al principi dels fitxers de les classes.

Per exemple:

```
import java.util.Scanner; 1

public class ScannerDemo {
    public static void main(String[] args) {
        Scanner in; 2
        in = new Scanner(System.in); 3
        System.out.print("a = ");
        int a = in.nextInt(); 4
        System.out.print("b = ");
        int b = in.nextInt(); 5
        int sumAB = a + b;
        System.out.print(a + " + " + b + " = " + sumAB);
    }
}
```

- ➊ El tipus de dades **Scanner** es troba en una llibreria externa anomenada **java.util** i cal importar-lo.
- ➋ Declarem una variable de tipus **Scanner** com si es tractés de qualsevol altre variable. A l'exemple li posem de nom **in** però podeu posar-li el nom que volgueu.
- ➌ Creem el contingut de la variable **in**. Ficeu-vos que **Scanner** comença en majúscula i per tant indica un tipus de dades que s'emmagatzema al **heap**, recordeu que per crear variables al **heap** cal utilitzar l'operador **new**. A més, enllacem el nou **Scanner** amb el flux d'entrada estàndard, **System.in**.
- ➍ A través de la variable **in**, que és de tipus **Scanner** demanem que ens extregui un **int** del flux d'entrada estàndard. Poden passar tres coses:
 - a. Si ja hi ha un **int** dins del flux, l'estreu i el retorna posant-lo a la variable **a**.
 - b. Si hi ha algun valor dins del flux que no es pot convertir a **int** es llençarà un error i s'interromprà l'execució del programa.
 - c. Si el flux d'entrada està buit, el programa pararà en espera que l'usuari introduceixi algun valor. ..

⑤ Exactament com el pas anterior

A l'exemple hem demanat dades de tipus **int** però **Scanner** ens permet demanar diferents tipus de dades. Per fer-ho es poden utilitzar els següents mètodes:

nextInt()

Retorna el següent valor de tipus **int** que s'introdueix per teclat.

nextLong()

Retorna el següent valor de tipus **long** que s'introdueix per teclat.

nextByte()

Retorna el següent valor de tipus **byte** que s'introdueix per teclat.

nextShort()

Retorna el següent valor de tipus **short** que s'introdueix per teclat.

nextDouble()

Retorna el següent valor de tipus **double** que s'introdueix per teclat.

nextFloat()

Retorna el següent valor de tipus **float** que s'introdueix per teclat.

nextBoolean()

Retorna el següent valor de tipus **boolean** que s'introdueix per teclat.

Els valors **true** o **false** es poden introduir en qualsevol combinació de majúscules i minúscules.

next()

Retorna una cadena consistent en els següents caràcters entrats per teclat fins al delimitador definit sense incloure el propi delimitador.

Per defecte aquest delimitador és l'espai.

nextLine()

Llegeix la resta de línia fins a trobar un **\n** i retorna la línia sencera sense el final de línia en format **String**.

useDelimiter(nou_delimitador)

Permet canviar el delimitador utilitzant la classe Scanner per separar els diferents valors.



Per defecte la classe **Scanner** determina on comença un valor i a on acaba delimitant-los amb espais en blanc, aquest comportament es pot modificar canviant els delimitadors amb `useDelimiter(nou_delimitador)`.

11.3. Problemes amb la classe Scanner

La classe **Scanner** pot tenir un comportament no esperat. Mira els exercicis per a més informació.

12

Programació estructurada

La programació estructurada és un **paradigma de programació** sorgit a la dècada de 1960, particularment del treball de Böhm i Jacopini, i un famós escrit de 1968: «Go To Statement Considered Harmful», de Edsger Dijkstra.

Els seus postulats es veurien reforçats, a nivell teòric, pel **teorema de la programació estructurada** i, a nivell pràctic, per l'aparició de llenguatges dotats d'estructures de control consistentes i ben formades.

12.1. Orígens de la programació estructurada

A la fi dels anys 1970 va sorgir una nova manera de programar que no solament permetia desenvolupar programes fiables i eficients, sinó que a més aquests estaven escrits de manera que es facilitava la seva comprensió en fases de millora posteriors.

El **teorema de la programació estructurada**, proposat per Böhm-Jacopini, demostra que **tot programa pot escriure's utilitzant únicament les tres instruccions de control següents:**

Sequència

Les instruccions ordenades s'executen una darrera l'altre, és a dir, seqüencialment.

Sel·lecció

Una o varies instruccions, un bloc d'instruccions, s'executen dependent de l'estat del programa.

Iteració

Una instrucció o un conjunt d'instruccions, un bloc d'instruccions, s'executen repetidament fins que el programa arriba a un estat determinat.

Taula 12.1. Representació gràfica de les tres estructures bàsiques de la programació estructurada.

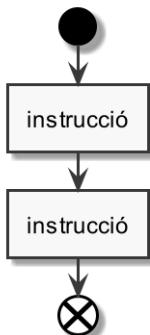


Figura 12.1. Sequència

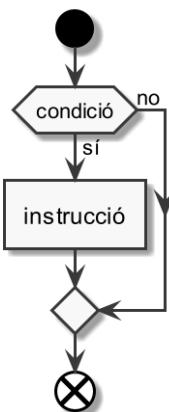


Figura 12.2. Sel·lecció

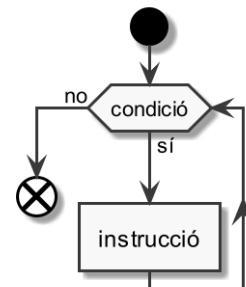


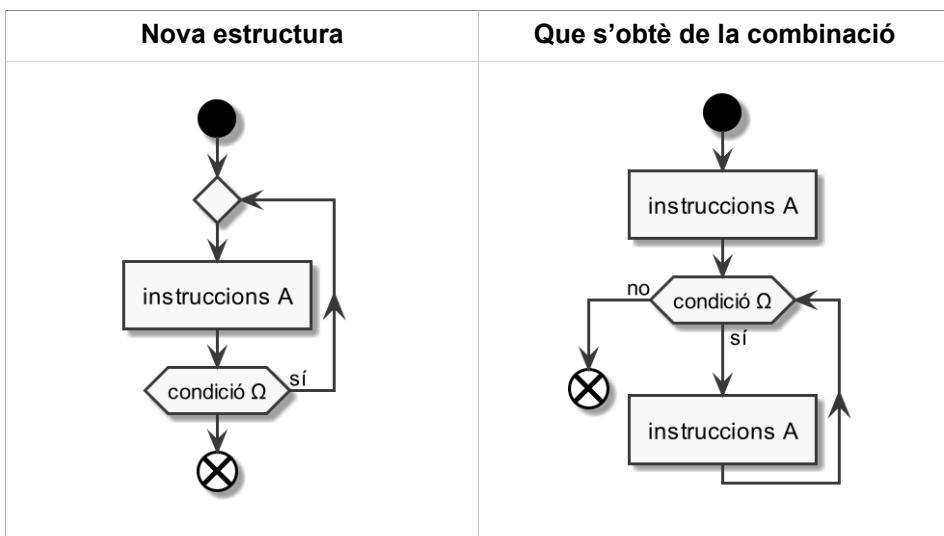
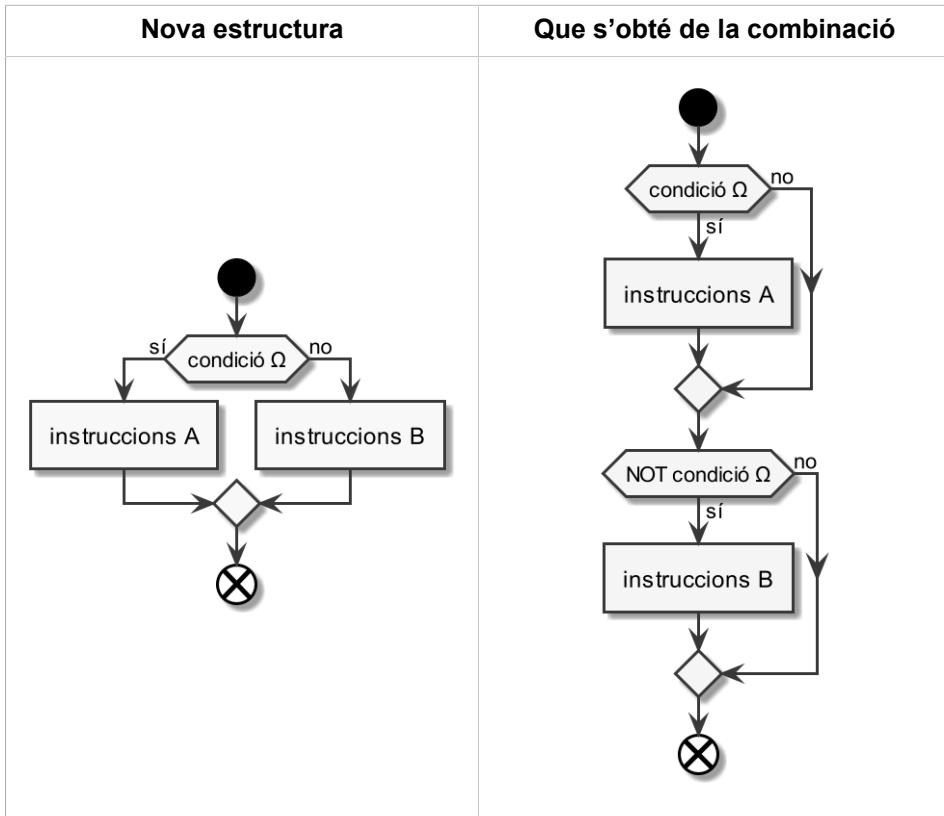
Figura 12.3. Repetició

Solament amb aquestes tres estructures es poden escriure tots els programes i aplicacions possibles. Si bé els llenguatges de programació tenen un major repertori d'estructures de control, aquestes poden ser construïdes mitjançant les tres bàsiques citades.

12.1.1. Estructures de control derivades

La majoria de llenguatges de programació que suporten el paradigma de programació estructurada accepten més estructures de control que les anteriors, no obstant, cal entendre que les noves estructures de control **sempre podran ser construïdes mitjançant les tres estructures bàsiques citades**

En general s'accepten les dues estructures següents:



12.2. Algorismes

Algorisme

En matemàtiques, lògica, ciències de la computació i disciplines relacionades, un algorisme **és un conjunt d'instruccions** o regles **definides i no-ambigües, ordenades i finites** que permet, típicament, **solucionar un problema**, realitzar un càlcul, processar dades i dur a terme altres tasques o activitats.

Donats un estat inicial i una entrada, seguint els passos successius s'arriba a un estat final i s'obté una solució. Els algorismes són l'objecte d'estudi de l'algorísmia.

En la vida quotidiana, s'empren algorismes sovint per a resoldre problemes. Alguns exemples són les **receptes de cuina** o les **instruccions sueques per muntar un móble**. Alguns exemples en matemàtiques són l'algorisme de multiplicació, per a calcular el producte, l'algorisme de la divisió per a calcular el quotient de dos números, l'algorisme d'Euclides per a obtenir el màxim comú divisor de dos enters positius, o el mètode de Gauss per a resoldre un sistema d'equacions lineals.

12.3. Mitjans d'expressió d'un algorisme

Els algorismes poden ser expressats de moltes maneres, incloent el **llenguatge natural, pseudocodi, diagrames de flux i llenguatges de programació** entre altres. Les descripcions en llenguatge natural tendeixen a ser ambigües i extenses, usar pseudocodi i diagrames de flux evita moltes ambigüïtats del llenguatge natural. Aquestes expressions són formes més estructurades per a representar algorismes; no obstant això, es mantenen independents d'un llenguatge de programació específic.

La descripció d'un algorisme usualment es fa a tres nivells:

Descripció d'alt nivell

S'estableix el problema, se selecciona un model matemàtic i s'explica l'algorisme de manera verbal, possiblement amb il·lustracions i ometent detalls.

Descripció formal

S'usa pseudocodi per a descriure la seqüència de passos que troben la solució.

Implementació

Es mostra l'algorisme expressat en un llenguatge de programació específic o algun objecte capaç de dur a terme instruccions.

També és possible incloure un teorema que **demonstre que l'algorisme és correcte**, un **anàlisi de complexitat** o tots dos.

12.4. Representació d'algoritmes

Per descriure un algoritme concret es poden utilitzar les següents tècniques.

12.4.1. Pseudocodi

El pseudocodi és un llenguatge informal d'alt nivell que usa les convencions i l'estructura d'un llenguatge de programació, però que està orientat a ser entès pels humans.

Exemple de pseudocodi 1.

```
algoritme Entrada
var
    i, k : enter
    x : real
    c: caràcter
fvar
inici
    i:= llegirEnter()
    x:= llegirReal()
    c:= llegirCaracter()
    k:= llegirEnter() + 7
falgoritme
```

Exemple de pseudocodi 2.

```

algoritme CondicionalsIDoble
    ...
    si condició llavors
        bloc_instruccions1
    altrament
        bloc_instruccions2
    fsi
    ...
falgoritme

```

Exemple de pseudocodi 3.

```

algoritme InstruccióMentre
    ...
    inicialitzacions
    mentre condició fer
        blocInstruccions1
        blocInstruccions2
        ...
        blocInstruccionsN
    fmentre
    blocInstruccionsN+1
    ...
falgoritme

```

12.5. Diagrama de control de flux

Consisteix en una subdivisió de passos seqüencials, d'acord amb les sentències i estructures de control d'un programa, que mostra els diferents camins que pot seguir un programa a l'hora d'executar les seves instruccions. Cada passa s'associa a una figura geomètrica específica.

També es pot representar l'algoritme directament en codi font d'un llenguatge de programació l'inconvenient principal d'aquest enfoc és que a diferència del pseudocodi i dels diagrames de flux l'interpret de l'algoritme ha de tenir coneixements informàtics per entendre'l.

Moltes vegades s'utilitzen les representacions gràfiques dels algoritmes per a establir una comunicació amb el client del programa que no té coneixements tècnics informàtics.

12.5.1. Alguns dels símbols utilitzats



i un exemple:

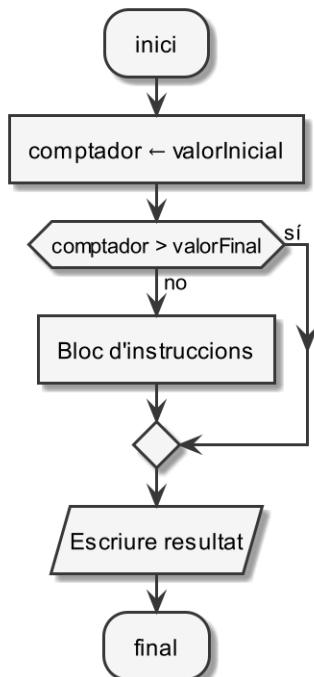


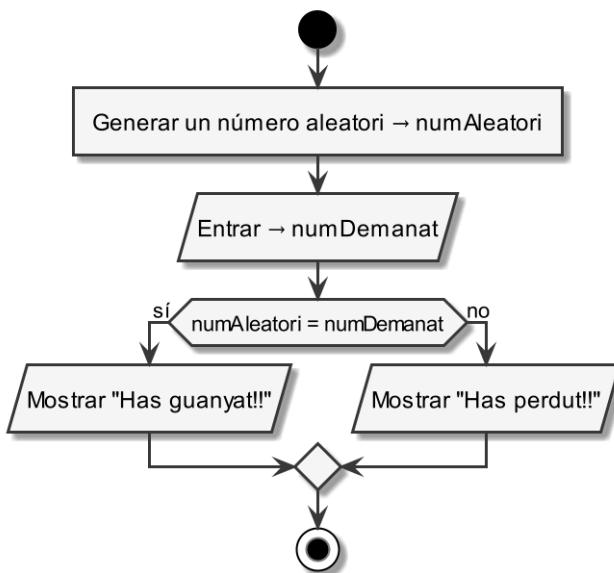
Figura 12.4. Exemple diagrama de control de flux

12.6. Alguns algoritmes d'exemple

12.6.1. Exemple 1

Escriure el diagrama de flux del següent programa:

- Generar un número aleatoriament.
- Demanar un número.
- Si els dos números coincideixen mostrar el missatge has guanyat, en cas contrari mostrar el missatge has perdut.



12.6.2. Exemple 2

Escriure el diagrama de flux del següent programa:

- Demanar un número enter, si el número entrat és senar tornar-lo a demanar, si el número és parell mostrar-lo i acabar el programa.

La següent solució té en compte únicament les estructures bàsiques de la programació estructurada:

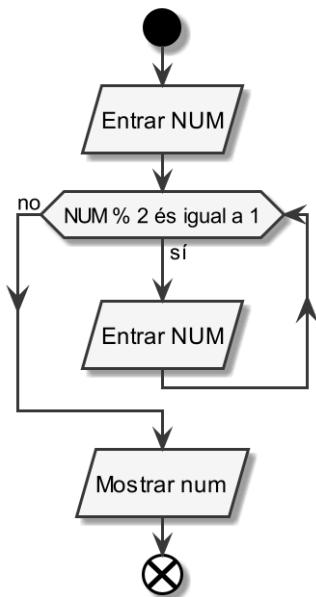


Figura 12.5. Primera proposta de solució

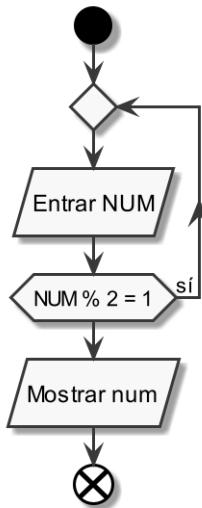
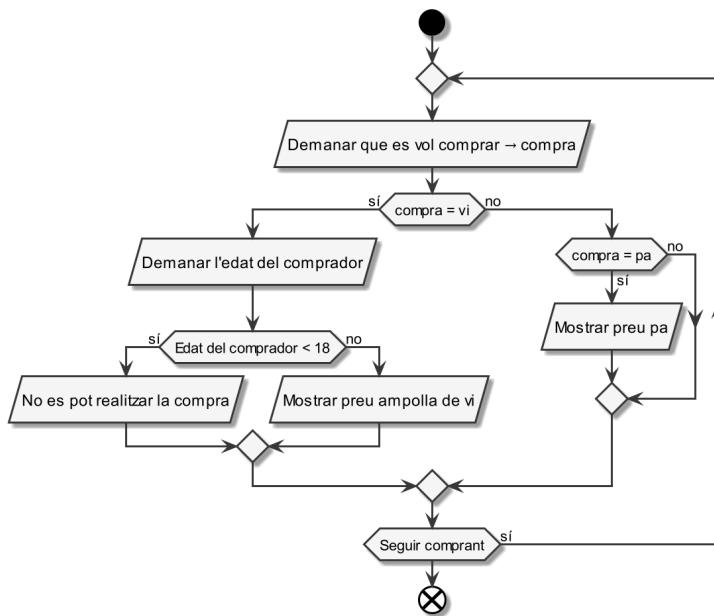


Figura 12.6. Segona proposta de solució

12.6.3. Exemple 3

Escriure el diagrama de flux del següent programa:

- Demanar que es vol comprar, una barra de pa o una ampolla de vi.
- Si es vol comprar una ampolla de vi demanar l'edat en anys del comprador.
- Si l'edat és menor que 18 indicar que no es pot realitzar la compra.
- En els altres casos mostrar el preu.
- Finalment demanar si es vol realitzar una altra compra i si es així tornar a començar el procés.



13

Estructures de selecció

13.1. Operadors relacionals

Els operadors relacionals s'utilitzen per comprovar diferents condicions com ara si dos valors són iguals o un és més gran que l'altre.

- El resultat obtingut amb un operador relacional és un valor de tipus **boolean**, és a dir, **true** o **false**.
- Els dos valors que relaciona un operador relacional han de ser compatibles.
- La majoria d'operadors relacionals **no funcionen amb Strings**.



Els operadors **==** i **!=** es poden utilitzar en Strings però **NO TENEN EL COMPORTAMENT ESPERAT !!!**. Per tant no els utilitzarem per a aquest tipus de dades.

- Per comprovar la igualtat entre cadenes s'utilitza el mètode **equals** de la classe String.

```
String fruta1 = "Poma";
String fruta2 = "Pera";
System.out.println(fruita1.equals(fruita2));
```

Taula 13.1. Llista dels operadors relacionals en Java

Operadors	Descripció	Tipus	Exemple	Resultat
==	igual que	binari	3 == 2	false

Operadors	Descripció	Tipus	Exemple	Resultat
equals	igual que [per tipus String]	binari	"hola".equals("hola")	
!=	no igual que	binari	3 != 2	true
>	major que	binari	3 > 2	true
>=	major o igual que	binari	3 >= 2	true
<	menor que	binari	3 < 2	false
#	menor o igual que	binari	3 # 2	false



Els 128 primers caràcters de la codificació unicode equivalen als de la codificació ASCII, això vol dir que el codi associat a cada caràcter creix alfabèticament.

No obstant els caràcters específics del català, accENTS, etc... no formen part dels primers 128 caràcters unicode.

Per tant es poden comparar caràcters sempre i quan es tingui present que s'estan comparant les codificacions numèriques unicode i no els caràcters com a tals.

13.2. Estructura if

La estructura **if** executa el següent bloc d'instruccions si i només si la condició proporcionada és veradera.

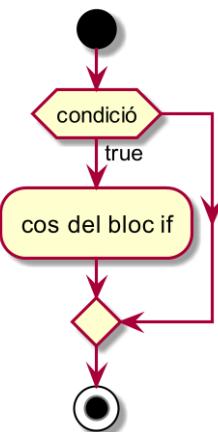


Figura 13.1. Estructura if

```
if (condició) {
    ...
}
```

Exemples:

```
if (x>0) {
    System.out.println("x és positiu");
}
```

Si el bloc d'instruccions a executar conté una sola instrucció no cal posar parèntesis.

```
if (x>0)
    System.out.println("x és positiu");
```

13.3. Estructura if-else

La estructura **if-else** decideix quin bloc d'instruccions executar en funció de si la condició proporcionada és veritadera o falsa.

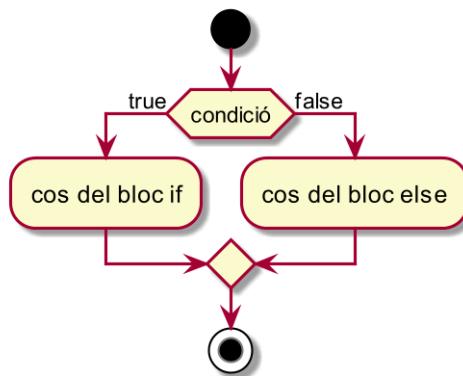


Figura 13.2. Estructura if-else

```
if (condició) {  
    ...  
} else {  
    ...  
}
```

Exemples:

```
if (x>0) {  
    System.out.println("x és positiu");  
} else {  
    System.out.println("x és negatiu");  
}
```

Colocació dels parèntesis

Hi ha dues maners habituals de sangrar i posar els parèntesis en una estructura if-else.

```
if (nombre % 2 == 0)
{
    System.out.println(nombre + " és parell");
}
else
{
    System.out.println(nombre + " és senar");
}
```

i

```
if (nombre % 2 == 0) {
    System.out.println(nombre + " és parell");
} else {
    System.out.println(nombre + " és senar");
}
```

13.4. Estructures if-else aniuades

Les estructures if-else es poden aniar per representar condicions més complexes.

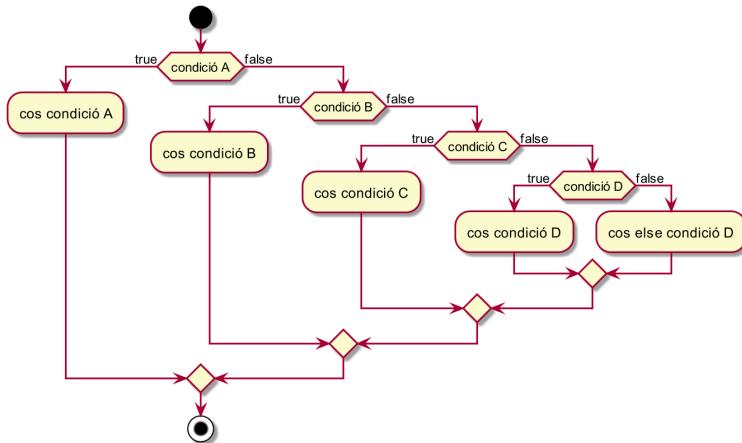


Figura 13.3. Estructures if-else aniuades

Per exemple:

```
if (x == 0) {
    System.out.println("x és zero");
} else {
    if (x > 0) {
        System.out.println("x és positiu");
    } else {
        System.out.println("x és negatiu");
    }
}
```

13.5. Estructures if-else encadenades

Les estructures if-else encadenades no representen un nou tipus d'estrucció en Java, es tracta simplement d'un conjunt de sentències if-else niuades. Però representen una idea diferent i es sangren de manera diferent per representar aquest fet.

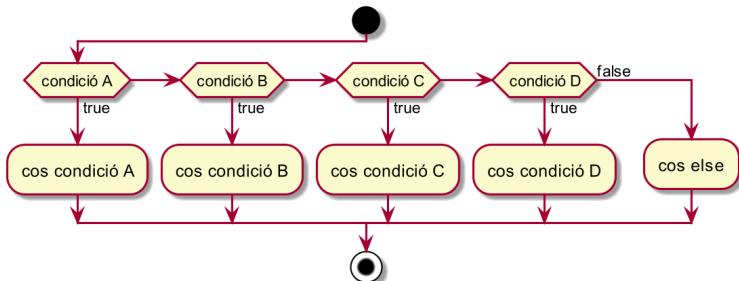


Figura 13.4. Estructrues if-else encadenades

Vegem-ho amb un exemple,

Volem fer un programa que en funció d'un valor entrat per l'usuari s'executi una opció del menú o una altra.

Una opció seria la següent:

```

if (valor == 1) {
    ...
} else {
    if (valor == 2) {
        ...
    } else {
        if (valor == 3) {
            ...
        } else {
            if (valor == 4) {
                ...
            }
        }
    }
}
  
```

Però és complicada d'entendre, té moltes sangries i és fàcil no saber un comencen i on acaben els blocs.

El codi anterior s'hauria d'escriure com:

```

if (valor == 1) {
    ...
} else if (valor == 2) {
    ...
}
  
```

```

} else if (valor == 3) {
    ....
} else if (valor == 4) {
    ....
}

```

En aquest cas la sangria deixa deixa molt clar la intenció del codi i és més fàcil de llegir.

13.6. Variables de tipus interruptor "flag variables"

Per emmagatzemar un valor *true* o *false* fa falta una variable de tipus **boolean**, es pot crear de la següent manera:

```

boolean resultatTest;
resultatTest = true;

```

Com que els operadors relacionals avaluen un valor booleà es pot guardar el resultat d'una comparació en na variable boolean.

```

boolean esPositiu = (x > 0); // Els parèntesis són innecessaris però
// faciliten la lectura del codi.

boolean esParell = (n % 2 == 0);

```

Es poden utilitzar variables de tipus boolean com a condició en una estructura condicional.

```

if (esPositiu)
    System.out.println("El valor era positiu en el moment de la
comprovació");

if (esParell) {
    System.out.println("El valor era parell en el moment de la
comprovació");
}

```

13.7. Operadors lògics

Els operadors lògics es poden utilitzar per crear expressions booleanes compostes.

Taula 13.2. Llista dels operadors lògics o booleans en Java

Operadors	Descripció	Tipus	Exemple	Resultat
!	negació	unari	!true	false
&&	and curtcircuit	binari	true & true	true
&	and	binari	true & false	false
	or curtcircuit	binari	true false	true
	or	binari	true false	false
^	xor [or exclusiu]	binari	true ^ true	false



La diferència entre & i && és que en el segon cas si la primera proposició és falsa ja no arriba a avaluar-se la segona.

La diferencia entre | i || es que si la primera proposición es cierta ya no arriba a evaluarse la segunda.

Exemples:

Exemple operador !

```
int i = 12;
boolean esNegatiu;
if (!esNegatiu) {
    System.out.println("i és postiu");
}
if (!(i < 10)) {
    System.out.println("i és major que 10");
}
```

Exemple operador &&.

```
int numerador = 10;
int denominador = 3;
if ((denominador != 0) && (numerador/denominador > 2)) {
    System.out.println("El quocient és positiu és major que 2");
}
```

En el cas anterior com que l'operador és && enlloc de & si el denominador fos 0 fallaria la primera condició de l'if i la segona ja no s'avaluaria.

Si l'operador fos & s'intentaria avaluar la segona condició encara que la primera hagués fallat, produint en aquest cas una divisió entre 0 i per tant un error d'execució ja que un valor d'infinít només es pot donar entre nombres de coma flotant.

Exemple operador ||.

```
float temperatura = 25f;
if ((temperatura <= 18) || (temperatura >= 29)) {
    System.out.println("La temperatura no és la adequada!!");
}
```

13.8. Estructura switch

Una expressió de tipus **switch** executa blocs de codi en base al valor d'una variable o expressió.

```
switch (expresión-del-switch) {
    case valor1:
        ....
    case valor2:
        ....
    ....
    case valorN:
        ....
    default:
        ....
}
```

- La expressió del switch s'ha d'avaluar com un tipus *byte, short, char, int, enum o String*.
- L'etiqueta **default** és opcional.

Funcionament de l'estructura switch:

- S'avalua l'expressió del switch.
- Si el valor de l'expressió del switch coincideix amb algun dels valors d'un case, l'execució comença des d'aquest punt i **s'executen totes les instruccions fins al final del switch**.

- Si el valor de l'expressió del switch no coincideix amb cap valor dels case, l'execució continua a partir de l'etiqueta default fins al final del switch.

Exemples:

Exemple 1.

```
int i = 10;
switch (i) {
    case 10: // Es troba coincidència de valors
        System.out.println("Deu"); // L'execució comença aquí.
    case 20:
        System.out.println("Vint");
    default:
        System.out.println ("No hi ha coincidència");
}
```

Deu
Vint
No hi ha coincidència

Exemple 2.

```
int i = 30;
switch (i) {
    case 10:
        System.out.println("Deu");
    case 20:
        System.out.println("Vint");
    default:
        System.out.println ("No hi ha coincidència"); // L'execució
        comença aquí.
}
```

No hi ha coincidència

Exemple 3.

```
int i = 30;
switch (i) {
    case 10: // Es troba coincidència de valors
        System.out.println("Deu");
```

```
default:  
    System.out.println ("No hi ha coincidència"); // L'execució  
començà aquí.  
case 20:  
    System.out.println("Vint");  
}
```

```
No hi ha coincidència  
Vint
```

13.9. Instrucció break

La instrucció **break** interromp l'execució d'una clàusula switch. L'execució del programa segueix a partir de la següent instrucció després del switch.

Exemple 1.

```
int i = 10;  
switch (i) {  
    case 10: // Es troba coincidència de valors  
        System.out.println("Deu"); // L'execució començà aquí.  
        break; // Transfereix l'execució fora de l'estruutura switch.  
    case 20:  
        System.out.println("Vint");  
        break;  
    default:  
        System.out.println ("No hi ha coincidència");  
        // No és necessari posar un break en aquest punt però és  
        recomana fer-ho  
        break;  
}
```

```
Deu
```

13.10. A vegades no utilitzar el break és útil

Tot i que en general l'ús de la instrucció *break* és necessari en algunes situacions és útil no utilitzar-lo.

Per exemple:

```

import java.util.Scanner;

public class JavaApplication69 {

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.println("Vols guardar el fitxer? [s/n]");

        String resposta = in.next(); // in.nextChar() no existeix!!

        switch (resposta) {
            case "s":
            case "S":
                // Guardem fitxer
                break;
            case "n":
            case "N":
                // NO guardem el fitxer
                break;
            default:
                // Tornem a mostrar la pregunta
                break;
        }
    }
}

```

13.11. Operador ternari ?:

L'operador ternari retorna un valor si determinada condició és veradera i retorna un altre valor si la mateixa condició és falsa.

La sintaxi de l'operador és:

condició ? valorSiFals : valorSiVertader

Per exemple:

Exemple 1.

```

import java.util.Scanner;

public class Major {

```

```
public static void main(String[] args) {  
    Scanner in = new Scanner(System.in);  
  
    System.out.print("Primer num: ");  
    int n1 = in.nextInt();  
    System.out.print("Segon num: ");  
    int n2 = in.nextInt();  
  
    System.out.printf("El major entre %d i %d és el %d%n", n1, n2,  
        (n1 > n2) ? n1 : n2);  
}  
}
```

Exemple 2.

```
import java.util.Scanner;  
  
public class Colors {  
  
    public static void main(String[] args) {  
        Scanner in = new Scanner(System.in);  
  
        System.out.println("Introdueix codi de color: \n[1] Vermell  
\n[2]Groc\n[3]Verd");  
        int codiColor = in.nextInt();  
  
        String colorText = (codiColor == 1) ? "Vermell" : ((codiColor  
== 2) ? "Groc" : ((codiColor == 3) ? "Verd" : "Desconegut"));  
        System.out.printf("Has triat el color %s%n", colorText);  
    }  
}
```

14

Estructures de repetició

Les **estructures de repetició, estructuras iterativas o bucles** permeten repetir una mateixa seqüència d'instruccions diverses vegades, mentre es compleixi una certa condició.

14.1. Estructura while

L'estructura **while** executa un bloc d'instruccions mentre una determinada condició sigui vertadera.

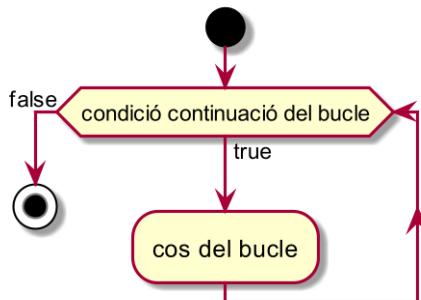


Figura 14.1. Estructura while

```
while (condició_continuació_del_bucle) {  
    // Cos del bucle  
    ....  
}
```

- La part del bucle que conté les instruccions que es repeteixen s'anomena **cos del bucle**.

- Cadascuna de les repeticions de les instruccions del bucle s'anomena **iteració**.
- L'estructura **while** conté una condició de continuació del bucle, una expressió boleana que s'avalua a cada iteració per determinar si s'executa el cos del bucle una alta vegada (condició **true**) o es finalitza l'execució del bucle (Condició **false**).



Cal assegurar que la condició del bucle és falsa en algun moment, en cas contrari es produirà un **bucle infinit**, és a dir, el bucle no acabarà mai.

Exemple 1.

```
int comptador = 0;
while (comptador < 100) {
    System.out.println("Iteració número: " + comptador);
    comptador++; // comptador + comptador + 1
}
```

```
Iteració número: 0
Iteració número: 1
....
...
Iteració número: 99
```

Exemple 2.

```
int suma = 0;
int i = 1;
while (i < 10) {
    suma = suma + i;
    i++;
}
System.out.println("La suma és " + suma); // sum = 45
```

Normalment, les variables de control dins d'un bucle es poden englobar dins d'algun d'aquests tipus de comportament:

Comptador

Una variable de tipus enter que va augmentant o disminuint, indicant de manera clara el nombre d'iteracions que caldrà fer.

Exemple de comptador.

```
// Mostra els 100 primers nombres parells
int comptador = 0;
while (comptador < 100) {
    System.out.println(comptador * 2);
    comptador++;
}
```

Acumulador

Una variable en què es van acumulant directament els càlculs que es volen fer, de manera que en assolir cert valor es considera que ja no cal fermés iteracions. Si bé s'assemblen als comptadors, no són ben bé el mateix.

Exemple d'acumulador.

```
// Mostra els múltiples de tres menors que 100
int valor = 3;
while (valor < 100) {
    System.out.print(valor + " ");
    valor += 3;
}
```

Semàfor (o flag)

Una variable que serveix com a interruptor explícit de si cal seguir fent iteracions. Quan ja no en volem fer més, el codi simplement s'encarrega d'assignar-li el valor específic que servirà perquè la condició avaluï *false*.

Exemple de semàfor.

```
boolean fiBucle = false;
while (fiBucle) {
    // en el moment en que es considera que s'ha de sortir del bucle
    // fiBucle = true;
}
```

14.2. Estructura do-while

L'estructura **do-while** és la mateixa que l'estructura **while** excepte que executa el cos del bucle en primer lloc i a continuació comprova la condició de finalització.

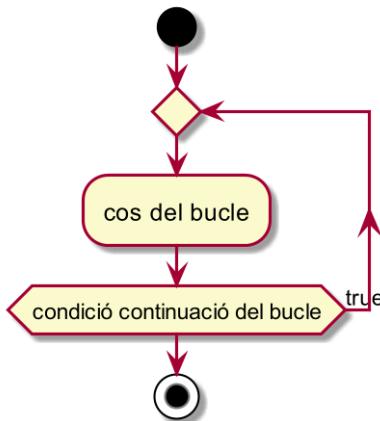


Figura 14.2. Estructura do-while

```

do {
    // Cos del bucle
    ...
} while (condició-continuació-del-bucle)
  
```



S'utilitza l'estructura **do-while** si les instruccions dins del cos del bucle s'han d'executar almenys una vegada.

Exemples:

```

// Calcula la suma dels enters que va entrant l'usuari
public static void main(String[] args) {
    int data;
    int sum = 0;

    Scanner input = new Scanner(System.in);

    // Seguim llegint enters fins que l'entrada no sigui 0
    do {
        System.out.print("Entra un enter (entra 0 per acabar): ");
        data = input.nextInt();

        sum += data;
    } while (data != 0);
  
```

```
System.out.println("La suma és " + sum);
}
```

14.3. Estructura for

L'estructura **for** proporciona una sintaxi compacte per aquells bucles en els que es coneix per avançat el nombre total d'iteracions a realitzar.

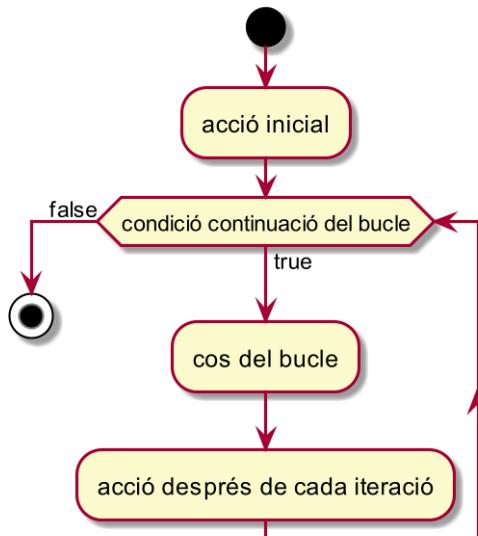


Figura 14.3. Estructura for

```
for (acció-inicial; condició-continuació-del-bucle; acció-després-de-
cada-iteració) {
    // Cos del bucle
}
```

Exemples:

Exemple 1.

```
// Mostrar la taula de multiplicar del 8
int i;
for (i = 1; i <= 10; i = i + 1) {
    System.out.printf("8 x %d = %d", i, 8 * i);
}
```



És una bona pràctica declarar la variable de control d'un bucle for dins de l'acció inicial del bucle. Només hauríem d'accedir a aquesta variable des de dins del bucle, mai des de fora.

El codi anterior quedaría:

```
// Mostrar la taula de multiplicar del 8
for (int i = 1; i <= 10; i = i + 1) {
    System.out.printf("8 x %d = %d", i, 8 * i);
}
```



Tant l'acció inicial com l'acció al final de cada iteració pot constar de més d'una instrucció. En aquest cas es separarien per comes.

Exemple.

```
for (int i = 0, int j = 0; i + j > 10; i++, j++) {
    // Fes alguna cosa
}
```



De manera similar a la instrucció **if**, els bucles **for** i els bucles **while** amb una sola instrucció no necessiten claus, no obstant aquesta pràctica denota un estil pobre. Per exemple, els següents fragments de codi són correctes, pot i que **no recomanats**.

```
while(a > 0)
    a--;
```

```
for(int i = 0; i <= 10; i++)
    System.out.println("i: " + i);
```

Com a conseqüència, els següents fragments de codi són sintàcticament correctes però generen **buckles infinitos**:

```
for (int i = 0; i < 5; i++);
```

```
while(a > 10);
```

14.4. Bucle "aniuats"

Un bucle pot estar dins d'un altre bucle.

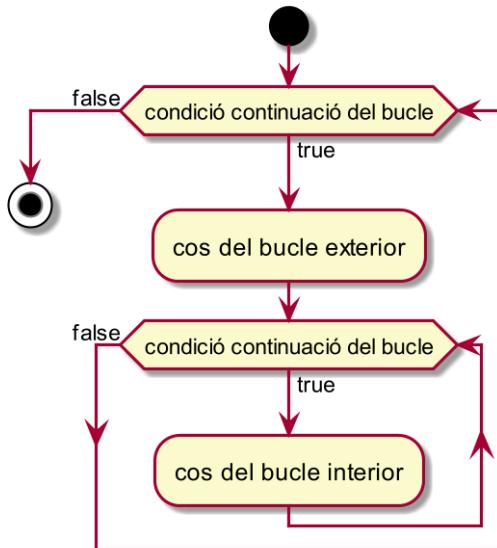


Figura 14.4. Estructura while aniuada

14.5. Instruccions break i continue

La instrucció **break** interromp l'execució d'un bucle. Quan s'executa un **break** el control es transfereix a la següent instrucció després del bucle.

Per exemple:

```
for(int num = 1; num < 100; num++) {
    System.out.println(num);
    if (num == 10) {
        break; // S'interromp el bucle quan num = 10;
    }
}
System.out.println("num: " + num); // 10
```

La instrucció **continue** salta una iteració dins d'un bucle.

Per exemple:

```
for (int i = 0; i < 10; i++) {  
    if (i % 2 == 0) {  
        continue;  
    }  
    System.out.println(i);  
}
```

14.6. Control de dades d'entrada amb la classe Scanner

La classe **Scanner** disposa d'una sèrie de mètodes que permeten controlar el tipus de dades que està introduint l'usuari.

Podem evitar que el programa deixi de funcionar si l'usuari entra un valor de tipus incorrecte.

Els següents mètodes retornen **true** si el següent valor recuperat per un objecte Scanner és del tipus demanat.

```
hasNextBoolean(), hasNextByte(), hasNextDouble(), hasNextFloat(),  
hasnextInt(), hasNextLine(), hasNextLong(), hasNextShort()
```

Exemple:

```
import java.util.Scanner;  
  
public class Exemple {  
    public static void main (String[] args) {  
        Scanner lector = new Scanner(System.in);  
  
        int valor = 0;  
        while (valor <= 0) {  
            System.out.print("Introdueix un valor enter? ");  
            if (lector.hasNextInt()) { // El següent valor és int,  
perfecte  
                valor = lector.nextInt();  
            } else { // El següent valor no és int  
                lector.next(); // i per tant saltem el valor  
incorrecte  
        }  
    }  
}
```

Control de dades d'entrada
amb la classe Scanner

```
        System.out.print("Aquest valor no és correcte. ");
    }
}
lector.nextLine();
}
```


15

Tipus de dades Compostes

Un **tipus de dada compost** és aquell que permet emmagatzemar més d'un valor dins d'una única variable.

15.1. Matrius

Una matriu és una estructura de dades de **longitud fixa** que permet emmagatzemar més d'un valor del **mateix tipus de dades**.

Les matrius s'utilitzen per emmagatzemar una col·lecció de dades del mateix tipus. Enlloc de declarar variables individuals, com per exemple, **nom1**, **nom2**, **nom3**, ..., **nom99** podríem declarar na matriu anomenada *noms* i utilitzar **noms[0]**, **noms[1]**, **noms[2]**, ..., **noms[99]** per representar les variables individuals.



A les matrius d'una sola dimensió se'ls sol anomenar **vectors**.

15.1.1. Declaració de variables de tipus Matriu

Les matrius són un tipus de dades en Java i per tant es poden declarar variables de tipus matriu.

- La sintaxis de declaració de les variables de tipus matriu és específica d'aquesta estructura de dades.

Sintaxis per la declaració de matrius:

```
tipus-de-dades-de-la-matriu[] nom-de-la-variable;
```

Exemples:

```
int[] numeros; // Declara una matriu de **int**s
String[] noms; // Declara una matriu de **String**s
```



Alternativament, les matrius es poden declarar posant els claudàtors després del nom de la variable enlloc de després del tipus. En general **es prefereixen els claudàtors després del tipus.**

Per tant són correctes:

```
int[] m; // és equivalent a
int m[];
```

Creació de matrius

Una matriu és un tipus per referència i per tant:

- La declaració de la variable no reserva espai a la memòria per emmagatzemar la matriu, només es reserva un espai per guardar la posició de memòria dins la que s'emmagatzemarà la matriu.
- Una variable matriu sense inicialitzar contindrà un valor especial anomenat **null**, en el moment en que s'inicialitzi contindrà l'adreça de memòria on es troba la matriu.
- Necessitarà l'operador **new** per a crear l'objecte.



Per crear un objecte matriu caldrà saber la seva longitud en el moment de la creació.

Un cop creada la matriu, la seva longitud no es pot modificar.

Sintaxis per a la creació d'una matriu:

```
nom-de-la-variable = new tipus-de-dades-de-la-matriu[longitud-de-la-
matriu];
```

Exemples:

```
int[] numeros; // Declaració de la matriu
numeros = new int[10]; // Creació d'una matriu de 10 elements de tipus
int

String noms[] = new String[100]; // Declaració i creació d'una matriu de
100 Strings
```

15.1.2. Assignació i lectura de valors d' una matriu



Els elements d'una matriu estan indexats a partir del zero.

Per tant una matriu m de 10 elements s'indexa de 0 a 9. ($m[0], m[1], \dots, m[9]$)

Accedir a un element fora del rang de la matriu donarà un error d'execució.

Assignació manual de valors a una matriu

Un cop disposem d'una matriu creada podem omplir-la de la següent manera:

```
int[] valors;
valors = new int[5];

valors[0] = 4;
valors[1] = 56;
valors[2] = 43;
valors[3] = 12;
valors[4] = 10;
```

Assignació manual de valors a una matriu amb l'ajuda d'un bucle

És fàcil omplir una matriu amb l'ajuda d'un bucle:

```

int[] valors;
valors = new int[5];

// Assignem a cada posició de la matriu la seva posició al quadrat.
// valor[0] = 0*0, valor[1] = 1*1, ....
for(int i = 0; i < 5; i++) {
    valors[i] = i*i;
}

```

Assignació de valors d'una matriu amb un inicialitzador

Els inicialitzadors permeten assignar valors concrets a les matrius en el moment de la seva declaració.

Per exemple:

```
char[] valors = {'a', 'b', 'c', 'd', 'e'};
```

equival a

```

char[] valors = new valors[5];

valors[0] = 'a';
valors[1] = 'b';
valors[2] = 'c';
valors[3] = 'd';
valors[4] = 'e';

```

Lectura de valors d'una matriu

Podem llegir els valors d'una matriu de la següent manera.

```

int[] valors = {2, 3, 4, 5};

for(int i = 0; i < 4; i++) {
    System.out.print("valors[" + i + "] = " + valors[i]);
}

```

15.1.3. Bucle for-each

Java suporta una construcció especial de bucle anomenada bucle **for-each** que permet recórrer una col·lecció, en particular una matriu, sense utilitzar una variable index.

El següent codi mostra tots els elements de la matriu *matriu*.

```
// matriu és una matriu d'elements double
for (double element: matriu) {
    System.out.println(element);
}
```

15.1.4. Longitud d'una matriu, propietat length

Accedir a un element d'una matriu fora de rang produeix un error de tipus **ArrayIndexOutOfBoundsException**.

La propietat **length** d'un objecte matriu permet saber la capacitat de la matriu.

Exemple:

```
int[] matriu = {1, 2, 3, 4, 5};

System.out.println("La matriu pot contenir " + matriu.length + " elements");

for (int i = 0; i < matriu.length - 1; i++) {
    System.out.println(matriu[i]);
}
```

15.2. Operacions habituals amb les matrius

Al treballar amb matrius apareixen una sèrie de necessitats de forma habitual.

- Cercar elements dins d'una matriu
- Clonar una matriu
- Canviar la mida d'una matriu
- Ordenar els elements dins d'una matriu
- Determinar si dues matrius són iguals

15.2.1. Cerca d'un element dins d'una matriu

Existeixen diferents algoritmes per tractar el problema de la cerca de valors dins d'una matriu, alguns d'ells necessiten preparar la matriu d'alguna manera, per exemple ordenar-ne els elements abans de fer la cerca.

Veurem la implementació més simple d'una cerca en una matriu, anomenada cerca directa.

L'algoritme consisteix en desplaçar-se seqüencialment pels elements de la matriu i anar comprovant si hi ha una coincidència.

```
public class Cerca {  
  
    public static void main(String[] args) {  
        float[] valors =  
            {2f, 5.5f, 9f, 10f, 4.9f, 8f, 8.5f, 7f, 6.6f, 5f, 9f, 7f};  
        float valorCercat = 10f;  
        boolean trobat = false; // true si s'ha trobat el valor  
        int posicio = 0; // Posició de la matriu on es fa la  
        comprovació  
  
        while ((posicio < valors.length) && (!trobat)) {  
            if (valors[posicio] == 10) {  
                trobat = true;  
            }  
            posicio = posicio + 1;  
        }  
  
        if (trobat) {  
            System.out.println("El valor " + valorCercat + " s'ha  
        trobat");  
        } else {  
            System.out.println(valorCercat + " no existeix a la  
        matriu");  
        }  
    }  
}
```

15.2.2. Clonar una matriu

Considerem el següent fragment de codi:

```
int m[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int n[];
```

Volem clonar la matriu *m* a la matriu *n*.



Fer *n* = *m*; no clona la matriu.

Recordeu que les matrius són tipus per referència i per tant una variable de tipus matriu no emmagatzema al matriu sencera, sinó que conté l'adreça de memòria on es troba l'objecte matriu, en Java això es coneix com una **referència**.

Per tant, l'assignació anterior modifica el contingut de *n* posant-hi el mateix que conté *m*.

m conté l'adreça de memòria on s'ubica la matriu, per tant amb l'assignació anterior *n* també conté l'adreça de memòria on es troba la matriu.

El punt és que de matriu en seguim tenint una, el que tenim són dues variables que en contenen la ubicació en memòria. Per tant modificar la matriu *m* o modificar la matriu *n* és exactament modificar la mateixa matriu.

Vegem-ho amb un exemple:

```
public class TestReferencies {

    public static void main(String[] args) {
        int[] m = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        int[] n = m;

        /* Modifiquem la matriu a través de n ,
           mirem la matriu a través de m
           i veurem els canvis.
        */
        n[0] = n[0] * 100;

        for (int i = 0; i < m.length; i++) {
            System.out.print(m[i] + " ");
        }
    }
}
```

```
    }
}
```

100	2	3	4	5	6	7	8	9	10
-----	---	---	---	---	---	---	---	---	----

Per clonar una matriu cal fer-ho element a element.

Exemple:

```
int[] sourceArray = {2, 3, 1, 5, 10};
int[] targetArray = new int[sourceArray.length];
for (int i = 0; i < sourceArray.length; i++) {
    targetArray[i] = sourceArray[i];
}
```



Si els elements de la matriu fossin elements per referència tornaríem a tenir problemes!!.

15.2.3. Canvi de mida

És impossible canviar de mida una matriu ja creada.

Per modificar la mida d'una matriu cal fer-ne una de nova amb la nova longitud i copiar els elements de l'antiga manualment.

Per exemple, suposem que tenim una matriu de 5 elements i la volem fer créixer fins a 10.

```
int[] matriu = {2, 3, 1, 5, 10};
int[] novaMatriu = new int[10]; // Nova dimensió
for (int i = 0; i < matriu.length; i++) {
    novaMatriu[i] = matriu[i];
} // Aquest bucle només omple els 5 primers elements de novaMatriu
```

15.2.4. Ordenació dels elements de la matriu

Per ordenar una seqüència de valors hi ha diversos algoritmes. Els més eficients són força complexos i es troben més enllà dels objectius d'aquest mòdul.

A mode d'exemple analitzarem un dels algoritmes menys eficients dedicat a l'ordenació anomenat **SelectionSort** o algoritme de selecció.

Algoritme: (Suposem una ordenació ascendent)

1. Buscar l'element més baix de la matriu
2. Un cop trobat intercanviar-lo amb l'element situat en primera posició.
En aquest punt tenim el menor element al principi de la matriu.
3. Buscar l'element més baix de la matriu però començant per la segona posició.
4. Un cop trobat intercanviar-lo amb l'element situat en segona posició.
En aquest punt tenim les dues primeres posicions amb els elements més petits.
5. Buscar l'element més baix de la matriu però començant per la tercera posició.
6.
7. Anem repetint el procés fins a arribar al penúltim element de la matriu.

Implementació:

```
public class SelectionSort {
    public static void main (String[] args) {
        float[] matriu = {5.5f, 9f, 2f, 10f, 4.9f};

        for (int i = 0; i < matriu.length - 1; i++) {
            for(int j = i + 1; j < matriu.length; j++) {
                //La posició tractada té un valor més alt que el de la cerca.
                Els intercanviem.
                if (matriu[i] > matriu[j]) {
                    float aux = matriu[i];
                    matriu[i] = matriu[j];
                    matriu[j] = aux;
                }
            }
        }

        System.out.print("L'array ordenat és: [");
        for (int i = 0; i < matriu.length; i++) {
            System.out.print(matriu[i] + " ");
        }
        System.out.println("]");
    }
}
```

```
}
```

15.2.5. Comprovar si dues matrius són iguals

Si tenim dues matrius **m1** i **m2** i es calcula **m1 == m2** estarem comprovant si la referència **m1** i la referència **m2** apunten al mateix objecte matriu, no estarem comprovant si el contingut de les matrius **m1** ni **m2** és el mateix.

Per exemple, el següent fragment de codi:

```
int[] m1 = {1, 2, 3};  
int[] m2 = {1, 2, 3};  
if (m1 == m2) {  
    System.out.println("Les matrius són iguals");  
} else {  
    System.out.println("Les matrius són diferents");  
}
```

Resulta en:

```
Les matrius són diferents
```

En aquest cas **m1 == m2** és **false**, ja que **m1** i **m2** representen dos objectes diferents a la memòria de l'ordinador i per tant contenen adreces de memòria diferents.

En canvi el següent fragment de codi:

```
int[] m1 = {1, 2, 3};  
int[] m2 = m1;  
if (m1 == m2) {  
    System.out.println("Les matrius són iguals");  
} else {  
    System.out.println("Les matrius són diferents");  
}
```

Resulta en:

```
Les matrius són iguals
```

En aquest cas **m1 == m2** és **true** ja que tant **m1** com **m2** són referències al mateix objecte matriu, en realitat de matriu només n'hi ha una però accessible des de dues variables.

Si es vol comprovar si **el contingut de les matrius és el mateix** tenim dues possibilitats.

1. Recórrer manualment les dues matrius i comparar els valors respectius.

```
int[] m1 = {1, 2, 3};
int[] m2 = {1, 2, 3};

boolean sonIguals = true;
for (int i = 0; i < m1.length; i++) {
    if (m1[i] != m2[i]) {
        sonIguals = false;
        break;
    }
}

if (sonIguals) {
    System.out.println("Les matrius són iguals");
} else {
    System.out.println("Les matrius són diferents");
}
```

2. utilitzar el mètode **equals** de la classe **Arrays**.

```
int[] m1 = {1, 2, 3};
int[] m2 = {1, 2, 3};
if (Arrays.equals(m1, m2)) {
    System.out.println("Les matrius són iguals");
} else {
    System.out.println("Les matrius són diferents");
}
```

15.3. Vectors de vectors

Donat que una matriu pot contenir elements d'un tipus de dades qualsevol res impedeix posar dins de cada element d'una matriu una altra matriu. De fet es

pot seguir així indefinidament, podríem posar una matriu de matrius dins de cada element de la matriu, etc...

En definitiva, podríem fer coses com la següent:

```
int[] vector1 = {1, 2, 3}; ①
int[] vector2 = {10, 20}; ②

int[][] matriu; ③
matriu = new int[2][]; ④
matriu[0] = vector1; ⑤
matriu[1] = vector2; ⑥
```

- ② Declarem dos vectors d'enters de la manera habitual.
- ③ Si volem un nou vector que tingui com a elements els dos vectors anteriors, és a dir, un vector de vectors podem declarar una nova variable **matriu** que és un vector [] de **int[]** vectors d'enters, és a dir declarem una variable de tipus **int[][]**.
- ④ Creem la nova variable **matriu** amb dos elements de tipus vector d'enters.
- ⑥ Assignem el contingut als dos elements del vector de vectors **matriu**.

La manera anterior de treballar és **poc habitual** i innecessàriament complicada, rarament ens interessa treballar amb vectors de vectors amb mides arbitràries, el cas més usual és treballar amb vectors de vectors on els vectors continguts **són tots de la mateixa mida**, en aquest cas podem pensar que estem treballant amb **una estructura tabular** i per tant una estructura fàcil de recórrer amb bucles niuats.

Per exemple:

```
int[] vector1 = {1, 2, 3};
int[] vector2 = {10, 20, 30};

int[][] matriu = new int[2][];
matriu[0] = vector1;
matriu[1] = vector2;

for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 3; j++) {
```

```

        System.out.print(matriu[i][j] + " ");
    }
    System.out.println();
}

```

```

1 2 3
10 20 30

```

15.3.1. Matrius multidimensionals

Seguint el raonament anterior podríem considerar vectors de vectors de vectors i així indefinidament.

Dimensió d'una matriu

Si tenim un sol parell de claudàtors, per exemple **int[]** parlem d'un **vector** o **d'una matriu de dimensió 1**.

Si tenim dos parells de claudàtors, per exemple **int[][]** parlem d'una **matriu de dimensió 2**, i la podem assimilar amb una taula.

Si tenim tres parells de claudàtors, per exemple **int[][][]** parlem d'una **matriu de dimensió 3**, i la podem assimilar amb un cub.

En general parlarem de **matriu de dimensió n** quan fem referència a estructures del tipus **int[]...n...[]**

Ens centrarem en el cas bidimensional, és a dir vectors de vectors.

Estenent les restriccions que compleixen les matrius d'una dimensió al cas de les matrius multidimensionals:

- En una mateixa dimensió, totes la matrius tindran exactament la mateixa mida.
- El tipus de dada que es desa a les posicions de la darrera dimensió serà el mateix per a tots.

15.3.2. Declarar matrius de dues dimensions

La sintaxi per declarar una matriu bidimensional és:

```
tipus-de-dades[][] nom-de-la-matriu;
```

Exemple:

```
int[][] matriu;
```

15.3.3. Creació de matrius bidimensionals

La sintaxi per crear una nova matriu bidimensional és:

```
nom-de-la-matriu = new tipus-de-dades[longitud-de-la-matriu-contenedora]
[longitud-de-les matrius-contingudes]
```

Per exemple:

Exemple 1.

```
int m[][] = new m[2][3];
m[0][0] = 1;
m[0][1] = 1;
m[0][2] = 1;

m[1][0] = 2;
m[1][1] = 2;
m[1][2] = 2;
```

Equivalentment,

Exemple 2.

```
int m[][] = {{1, 1, 1}, {2, 2, 2}};
```



Podem pensar que una matriu bidimensional com una taula, la primera dimensió especifica el nombre de files de la taula i la segona dimensió especifica el nombre de columnes.

Per exemple, la següent declaració defineix una matriu de 5 files i 10 columnes.

```
int[][] m = new int[5][10];
```

Finalment, podem recórrer la matriu amb dos bucles i mostrar-ne el contingut:

```
for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 3; j++) {
        System.out.print(m[i][j]) + " ";
    }
    System.out.println(); // Forcem un salt de línia
}
```

```
1 1 1
2 2 2
```

Un altre exemple:

```
char[] symbols = {'*', '+', 'o'};
char[][] m = new char[4][5];

for (int f = 0; f < 4; f++) {
    for (int c = 0; c < 5; c++) {
        m[f][c] = symbols[(f + c) % 3];
    }
}

for (int f = 0; f < 4; f++) {
    for (int c = 0; c < 5; c++) {
        System.out.print(m[f][c] + " ");
    }
    System.out.println();
}
```

```
* + o * +
+ o * + o
o * + o *
* + o * +
```

15.4. Registres

En programació, un **registre** és un tipus de dades estructurat format per la unió de varis elements dins de la mateixa estructura. Aquests elements poden ser dades de tipus primitius o bé altres estructures de dades. Cadascun d'aquests elements s'anomena **camp**.



Treballar amb registres en Java és una mica artificial, Java treballa amb una estructura més elaborada que un registre anomenada **classe**.

Les classes tenen una sèrie de característiques de les que no disposen els registres, de fet, es pot considerar que un registre és una versió incompleta d'una classe, el que justifica que no es consideri un **bon estil** treballar amb registres en Java.

No obstant, el concepte de registre **si que és important** en programació i per tant utilitzarem aquest tipus de construcció deixant de banda la necessitat o no que té Java d'aquesta estructura.

15.4.1. Declarar l'estructura del registre

En primer lloc caldrà **declarar l'estructura del registre**, això significa explicitar quins camps componen el registre i quins tipus tenen cadascun d'ells. Per fer-ho s'utilitza la següent sintaxi:

```
class NomRegistre { ①
    tipus nomCamp1;
    tipus nomCamp2;
    tipus nomCamp3;
    ...
}
```

- ① El nom del registre sempre comença amb majúscula.

Vegem alguns exemples:

```
class Punt2D {
```

Crear nous registres a partir
de l'estructura del registre

```
double x;  
double y;  
}
```

```
class Client {  
    String dni;  
    String nom;  
    String primerCognom;  
    String segonCognom;  
    int edat;  
}
```

Fins i tot podríem definir estructures més complexes a partir d'estructures més simples, per exemple:

```
class Rectangle {  
    Punt2D origen;  
    Punt2D[] vertexs;  
}
```

15.4.2. Crear nous registres a partir de l'estructura del registre

Un cop declarada una estructura podem crear tants registres com faci falta a partir d'aquesta. El truc és pensar que una estructura **defineix un nou tipus de dades per referència** i per tant crear un nou registre consisteix en **crear una nova variable del nou tipus definit**.

Prenent com a referència els exemples anteriors:

Exemple 1.

```
public class Registres {  
  
    public static void main(String[] args) {  
        Punt2D p1 = new Punt2D();  
        p1.x = 10;  
        p1.y = 10;  
        Punt2D p2 = new Punt2D();  
        p2.x = 20;  
        p2.y = 20;
```

```
    double distancia = Math.sqrt(Math.pow(p2.x - p1.x, 2) +  
Math.pow(p2.y - p1.y, 2)); // Pitàgores  
    System.out.printf("La distància entre (%.2f, %.2f) i (%.2f,  
%.2f) és %.2f%n", p1.x, p1.y, p2.x, p2.y, distancia);  
}  
}  
  
class Punt2D {  
  
    double x;  
    double y;  
}
```

La distància entre (10,00, 10,00) i (20,00, 20,00) és 14,14

Exemple 2.

```
public class Registres {  
  
    public static void main(String[] args) {  
        Client[] clients = new Client[2];  
  
        clients[0] = new Client();  
        clients[0].dni = "39918453T";  
        clients[0].nom = "Rosa";  
        clients[0].primerCognom = "Teixido";  
        clients[0].segonCognom = "Valenciano";  
        clients[0].edat = 32;  
  
        clients[1] = new Client();  
        clients[1].dni = "89123996A";  
        clients[1].nom = "Vicente";  
        clients[1].primerCognom = "Mateo";  
        clients[1].segonCognom = "Moure";  
        clients[1].edat = 56;  
    }  
}  
  
class Client {  
  
    String dni;  
    String nom;
```

```

String primerCognom;
String segonCognom;
int edat;
}

```

15.4.3. Valors per defecte

Quan creem una nova instància d'un registre els camps d'aquests ja tenen uns valors per defecte, per exemple:

```

public class Registres {

    public static void main(String[] args) {
        Punt2D p = new Punt2D();

        System.out.printf("(%.2f, %.2f)%n", p.x, p.y);
    }
}

class Punt2D {
    double x;
    double y;
}

```

(0,00, 0,00)

Podem modificar els valors que s'estableixen per defecte quan es creen nous registres a partir del tipus de dades assignant el valor per defecte directament a cada camp a la pròpia definició del tipus, per exemple:

```

public class Registres {

    public static void main(String[] args) {
        Punt2D p = new Punt2D();

        System.out.printf("(%.2f, %.2f)%n", p.x, p.y);
    }
}

class Punt2D {

```

```
double x = 100;  
double y = 100;  
}
```

```
(100,00, 100,00)
```

Bibliografia

Llibres

Allen B. Downey. 'Think Java'. O'Reilly Media. 2016. ISBN 978-1-491-92956-8

Kishori Sharan. 'Beginning Java 8 Fundamentals'. Apress. 2014. ISBN 978-1-430-26653-2.

Daniel Liang. 'Introduction to Java Programming, Comprehensive Version'. 9th Edition. Pearson. 2012. ISBN 978-0-132-93652-1

Robert Liguori. 'Java 8 Pocket Guide'. O'Reilly Media. 2014. ISBN 978-1-491-90086-4

Stuart Reges. 'Building Java Programs: A Back to Basics Approach' 2nd Edition. Pearson. 2010. ISBN 978-0-136-09181-3

Walter J. Savitch. 'Java: An Introduction to Problem Solving & Programming'. 6th Edition. Pearson. 2011. ISBN 978-0-132-16270-8

Part II. UF2 Disseny modular

Controlling complexity is the essence of computer programming.

— Brian W. Kernighan *científic de la computació coautor del llibre "el llenguatge de programació C"*

Debugging is twice as hard as writing the code in the first place.
Therefore, if you write the code as cleverly as possible, you are,
by definition, not smart enough to debug it.

— Brian W. Kernighan *científic de la computació coautor del llibre "el llenguatge de programació C"*

Sumari

16. Resultats d'aprenentatge i criteris d'avaluació	135
17. Continguts	137
18. Tractament de cadenes	139
18.1. El tipus de dades String	139
18.1.1. Declaració i inicialització d'un objecte	140
18.1.2. Declaració i inicialització d'un objecte String	140
18.1.3. Els Strings són immutables	141
18.1.4. Combinar assignacions i sumes de cadenes pot ser molt inefficient	141
18.1.5. Els Strings no són adequats per fer experiments amb les referències	142
18.2. Mètodes i propietats de la classe String	142
18.3. Com s'interpreta una classe en UML	142
18.4. Mètodes de la classe String	145
18.4.1. equals(s1 : String): boolean	145
18.4.2. equalsIgnoreCase(s1: String): boolean	146
18.4.3. compareTo(s1: String): int	147
18.4.4. compareTolgnoreCase(s1: String): int	147
18.4.5. length(): int	148
18.4.6. charAt(index: int): char	148
18.4.7. substring(beginIndex: int): String	148
18.4.8. substring(beginIndex: int, endIndex: int): String	149
18.4.9. toLowerCase(): String	149
18.4.10. toUpperCase(): String	149
18.4.11. trim(): String	149
18.4.12. replace(oldChar: char, newChar: char): String	150
18.4.13. replaceFirst(oldString: String, newString: String): String	150
18.4.14. replaceAll(oldString: String, newString: String): String ...	150
18.4.15. split(delimiter: String): String[]	151
18.4.16. indexOf(ch: char): int	151
18.4.17. indexOf(ch: char, fromIndex: int): int	151
18.4.18. indexOf(s: String): int	151
18.4.19. indexOf(s: String, fromIndex: int): int	151
18.4.20. lastIndexOf(ch: int): int	152

18.4.21. lastIndexOf(ch: int, fromIndex: int): int	152
18.4.22. lastIndexOf(s: String): int	152
18.4.23. lastIndexOf(s: String, fromIndex: int): int	152
18.4.24. format(format, item1, item2, ..., itemk): String	152
18.5. Conversió entre cadenes i altres tipus de dades	152
18.5.1. toCharArray(): char[]	153
18.5.2. valueOf(...): String	153
18.6. Classes embolcall o Wrapper Classes	154
18.7. Conversió entre cadenes i números	155
18.8. El tipus de dades Character	155
18.9. Mètodes de la classe Character	156
18.10. Lectura de cadenes amb la classe Scanner	157
18.11. Cadenes i la configuració regional	158
18.12. Classes StringBuilder i StringBuffer	159
19. Mètodes (funcions)	161
19.1. Definir un mètode	162
19.2. Paraula clau return	163
19.3. Cridar un mètode	164
19.4. Ubicació dels mètodes	165
19.5. La pila de crides a funcions	166
19.6. Passar arguments per valor	168
19.7. Referències passades per valor	172
19.8. Passar matrius als mètodes	173
19.9. Tornar una matriu des d'un mètode	177
19.10. Arguments per la línia de comandes	177
19.11. Sobrecàrrega de mètodes	178
19.12. Àmbit de les variables	180
19.12.1. Variables d'àmbit de classe	180
19.12.2. Constants d'àmbit de classe	181
19.12.3. Variables d'àmbit de mètode	181
19.12.4. Variables d'àmbit de bloc	182
19.12.5. "shadowing"	183
19.13. Llista d'arguments de longitud variable	184
20. Programació modular en Java	187
20.1. Mètodes / funcions	187
20.2. Classes estàtiques	188

20.3. Biblioteques	189
20.3.1. Declaració de mòduls	189
20.3.2. Declaració de paquets	189
20.3.3. El paquet per defecte	191
20.4. Ubicació de les classes en paquets	192
20.4.1. Com ho fa Java per trobar les classes definides per l'usuari	192
20.4.2. Declaracions import	195
20.5. Fitxers jar	196
20.6. Les biblioteques del llenguatge java	198
20.6.1. Bilblioteques abans i després de Java8	198
20.6.2. Moduls de Java11	198
20.6.3. Biblioteques de Java8	199
20.6.4. java.lang	199
20.6.5. java.util	200
20.6.6. java.util.Arrays	200
20.6.7. java.util.ArrayList	202
20.6.8. java.time	204
20.6.9. java.io	205
20.6.10. java.net	205
20.6.11. java.security	205
20.6.12. java.sql	205
20.6.13. java.swing	205
20.6.14. javafx	205
21. Alguns problemes difícils	207
21.1. Implementar l'eina "rellenar"	207
21.2. Realitzar cerques en una estructura de directoris	210
22. Recursivitat	213
22.1. Implementació interna en un ordinador	213
22.2. Algoritmes recursius	214
22.3. Exemple - comptaEnrrera	214
22.4. Exemple - mostrarLlínies	215
22.5. Exemple - Càcul del factorial d'un número	215
22.6. Exemple - Successió de Fibonacci	216
22.7. Exemple - Palindroms	217
22.8. Exemple - cercaBinaria	218

22.9. Conveniència de l'ús de la recursividat	219
22.10. Backtracking	219
23. Introducció a Processing	221
23.1. Mètode setup()	221
23.2. Mètode draw()	221
23.3. Paràmetres típics d'inicialització	221
23.4. Coordenades	222
23.5. Formes bàsiques	222
23.6. Mètode <i>keyPressed()</i>	223
23.7. Mètode <i>mousePressed()</i>	224
23.8. Mostrar text	225
23.9. Mostrar imatges	226
23.10. Temps	228
23.11. Animacions	229
23.12. Control de col·lisions	230
Bibliografia	233

16

Resultats d'aprenentatge i criteris d'avaluació

1. Escriu i prova programes senzills reconeixent i aplicant els fonaments de la programació modular.
 1. Analitza els conceptes relacionats amb la programació modular.
 2. Analitza els avantatges i la necessitat de la programació modular.
 3. Aplica el concepte d'anàlisi descendent en l'elaboració de programes.
 4. Modula correctament els programes realitzats.
 5. Realitza correctament les crides a funcions i la seva parametrització.
 6. Té en compte l'àmbit de les variables en les crides a les funcions.
 7. Prova, depura, comenta i documenta els programes.
 8. Defineix el concepte de llibreries i la seva utilitat.
 9. Utilitza llibreries en l'elaboració de programes.
10. Coneix les nocions bàsiques de la recursivitat i les seves aplicacions clàssiques.

Continguts

1. Programació modular:
 1. Concepte.
 2. Avantatges i inconvenients.
 3. Anàlisi descendant (top down).
 4. Modulació de programes.
 5. Crides a funcions. Tipus i funcionament.
 6. Àmbit de les crides a funcions.
 7. Prova, depuració i comentaris de programes.
 8. Concepte de llibreries.
 9. Ús de llibreries.
10. Introducció al concepte de recursivitat.

18

Tractament de cadenes

Una cadena de text és la representació d'un conjunt de caràcters tractats com un conjunt.

18.1. El tipus de dades String

El tipus de dades **String** és un tipus de dades per referència que serveix per representar i gestionar de manera senzilla una seqüència de caràcters.

En el llenguatge d'orientació a objectes un tipus de dades per referència s'anomena una **classe**.



Recordeu que un **tipus de dades per referència** és un tipus de dades tal que les variables generades a partir d'aquest tipus s'emmagatzemen al "heap", en contraposició als **tipus de dades primitius** que emmagatzemen les seves variables al "stack".

Podeu llegir una descripció més detallada de la diferència entre aquests tipus de dades a l'apartat [Diferències entre tipus primitius i tipus per referència](#).

Les variables creades a partir d'un tipus per referència contenen **objectes** que són representacions concretes d'una determinada classe.

Per exemple:

```
Scanner in = new Scanner(System.in);
```

1. **Scanner** és un tipus de dades i per tant una **classe**.
2. **in** és una variable de tipus **Scanner** i per tant és una referència a un objecte de tipus **Scanner** i per tant un **objecte**.



En java els noms dels tipus de dades per referència, és a dir, de les classes comencen en majúscula.

Per exemple:

String, Scanner.

18.1.1. Declaració i inicialització d'un objecte

Les variables de tipus per referència contenen l'adreça de memòria on s'ubica l'objecte al que fan referència.

La declaració d'aquestes variables és idèntica a la declaració de variables de tipus primitius.

La inicialització no ho és,

- en primer lloc caldrà trobar una ubicació contigua de memòria amb prou capacitat per ubicar el nou objecte.
- i en segon lloc caldrà crear l'objecte obtenir l'adreça de memòria on s'ubica i modificar el contingut de la variable a aquesta adreça.

Tot això és fa amb l'ajuda de l'operador **new**.

Per exemple:

```
Scanner in; // Creem la variable per mantenir la referència al nou
            // objecte.
in = new Scanner(); // Busquem un espai de memòria adient per
                  // emmagatzemar un nou Scanner, retornem l'adreça de memòria on es troba i
                  // la guardem a la variable in.
```

18.1.2. Declaració i inicialització d'un objecte String

Pel comentat anteriorment una variable de tipus **String** s'hauria de crear de la següent manera:

```
String s;
s = new String("Hola");
```

No obstant el tipus de dades **String** permet declarar i crear les variables de la mateixa manera que els tipus per primitius.

```
String s;
s = "Hola";
```



Encara que les variables de tipus **String** es puguin crear com si es tractés d'un tipus primitiu cal tenir present que **String segueix essent un tipus per referència**.

18.1.3. Els Strings són immutables

Un objecte de tipus **String** és **immutable**, un cop creat no es pot canviar.

El següent fragment de codi no canvia el contingut d'una cadena sinó que crea dues cadenes i manté una referència a la segona.

```
String s = "Hola";
s = "Adéu";
```

18.1.4. Combinar assignacions i sumes de cadenes pot ser molt ineficient

El fet que els **String** siguin immutables implica que generar noves cadenes a partir de la concatenació de cadenes existents sigui molt poc eficient ja que **cada cop que es realitza l'assignació d'una suma de cadenes es crea una nova cadena amb el contingut de les cadenes sumands**.

Per exemple, el següent codi és extremadament ineficient:

```
public class Cadenes {

    public static void main(String[] args) {
        String s = "1";

        for (int i = 0; i < 100000; i++) {
```

Els Strings no són adequats per fer experiments amb les referències

```
s += ", " + i;  
}  
}  
}
```

18.1.5. Els Strings no són adequats per fer experiments amb les referències

Com que els Strings són immutables i s'utilitzen molt, la màquina virtual de Java utilitza una única instància pels literals de tipus cadena que tenen la mateixa seqüència de caràcters amb l'objectiu de millorar l'eficiència.

Aquest comportament és exclusiu del tipus **String** i pot ser font de confusió si s'utilitza aquest tipus per fer proves amb els tipus per referència.

Per exemple:

```
String s1 = "Hola"; // Literal de tipus cadena  
String s2 = new String("Hola"); // No és un literal!!, és un objecte  
cadena  
String s3 = "Hola";  
  
System.out.println(s1 == s2); // false  
System.out.println(s1 == s3); // true, són exactament el mateix objecte
```

18.2. Mètodes i propietats de la classe String

La classe **String** proporciona un conjunt de mètodes i propietats pel tractament de cadenes.

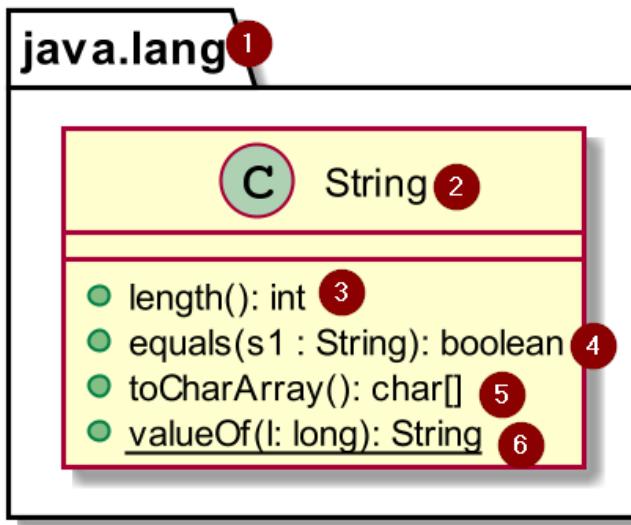
S'hi accedeix a partir d'un objecte de tipus String mitjançant l'operador .

Per exemple:

```
String s = "Hola"; // s és un objecte de tipus literal de cadena  
s.metode1(); // Accedeix al mètode metode1 dins de l'objecte s.
```

18.3. Com s'interpreta una classe en UML

La notació utilitzada per representar els diferents mètodes de les classes és la de UML, un llenguatge de modelat molt utilitzat.



1. Nom del paquet on es troba la classe.
2. El quadre fa referència a una **Classe**.
3. El nom del tipus de dades (o el nom de la classe) és **String**.
4. **length** és un mètode sense paràmetres d'entrada i retorna un int.

```
String s = "cadena";
int l = s.length();
```

5. **equals** és un mètode amb un paràmetre d'entrada de tipus String i retorna un **boolean**.

```
String s = "cadena";
boolean b = s.equals("una otra cadena");
```

6. **toCharArray** és un mètode sense paràmetres d'entrada i retorna una matriu de char.

```
String s = "cadena";
char[] matriu = s.toCharArray();
```

7. Aquest mètode està **subratllat**, això vol dir que **no es crida a través d'un objecte String, sinó que es crida a través de la pròpia classe**.

El següent exemple compila però és incorrecte:

```
String s = "cadena";
long l = 10l;

String resultat = s.valueOf(l); // No hauria de cridar el mètode des
d'un a variable String
```

i la versió correcte:

```
long l = 10L;

String resultat = String.valueOf(l);
```

18.4. Mètodes de la classe String

java.lang

C String

- equals(s1 : String): boolean
- equalsIgnoreCase(s1: String): boolean
- compareTo(s1: String): int
- compareToIgnoreCase(s1: String): int
- length(): int
- charAt(index: int): char
- substring(beginIndex: int): String
- substring(beginIndex: int, endIndex: int): String
- toLowerCase(): String
- toUpperCase(): String
- trim(): String
- replace(oldChar: char, new Char: char): String
- replaceFirst(oldString: String, new String: String): String
- replaceAll(oldString: String, new String: String): String
- split(delimiter: String): String[]
- indexOf(ch: char): int
- indexOf(ch: char, fromIndex: int): int
- indexOf(s: String): int
- indexOf(s: String, fromIndex: int): int
- lastIndexOf(ch: int): int
- lastIndexOf(ch: int, fromIndex: int): int
- lastIndexOf(s: String): int
- lastIndexOf(s: String, fromIndex: int): int
- toCharArray(): char[]
- format(format, item1, item2, ..., itemk): String
- valueOf(c: char): String
- valueOf(data: char[]): String
- valueOf(d: double): String
- valueOf(f: float): String
- valueOf(i: int): String
- valueOf(l: long): String
- valueOf(b: boolean): String

18.4.1. equals(s1 : String): boolean

Retorna true si el **contingut** de la cadena sobre la que es crida **equals()** és igual que **s1**.

Exemple:

```
String s1 = "Hola";
String s2 = "hola";

System.out.println("Es s1 igual a s2? " + s1.equals(s2));
```

És s1 igual a s2? false



No confondre el mètode **equals()** amb l'operador **==**, el primer comprova **si el contingut** de dues cadenes és el mateix, el segon comprova **si les dues cadenes són la mateixa**, és a dir, si ocupen la mateixa adreça de memòria. Per exemple:

```
String s1 = new String("abc");
String s2 = new string("abc");

System.out.println(s1.equals(s2)); // true
System.out.println(s1 == s2); // false
```

Per altra banda,

```
String s1 = new String("abc");
String s2 = s1;

System.out.println(s1.equals(s2)); // true
System.out.println(s1 == s2); // true
```

18.4.2. **equalsIgnoreCase(s1: String): boolean**

Retorna **true** si la cadena sobre la que es crida el mètode és igual que **s1** sense tenir en compte majúscules i minúscules.

```
String s1 = "Hola";
String s2 = "hola";

System.out.println("Es s1 igual a s2? " + s1.equalsIgnoreCase(s2));
```

És s1 igual a s2? true

18.4.3. *compareTo(s1: String): int*

Retorna un enter major que 0, igual a 0 o menor que 0 per indicar si la cadena és major, igual o menor que s1.

El criteri de comparació és lexicogràfic, és a dir, en termes de la ordenació Unicode.

```
String s1 = "abc";
String s2 = "bcd";

if (s1.compareTo(s2) < 0) {
    System.out.printf("%s és més gran que %s %n", s2, s1);
} else if (s1.compareTo(s2) > 0) {
    System.out.printf("%s és més gran que %s %n", s1, s2);
} else {
    System.out.println("Les dues cadenes són iguals");
}
```

bcd és més gran que abc

18.4.4. *compareToIgnoreCase(s1: String): int*

Retorna un enter major que 0, igual a 0 o major que 0 per indicar si la cadena és major, igual o menor que s1.

El criteri de comparació és lexicogràfic, és a dir, en termes de la ordenació Unicode. A més **no té en compte les majúscules o minúscules**.

```
String s1 = "abc";
String s2 = "ABC";

if (s1.compareTo(s2) < 0) {
    System.out.printf("%s és més gran que %s %n", s2, s1);
} else if (s1.compareTo(s2) > 0) {
    System.out.printf("%s és més gran que %s %n", s1, s2);
} else {
    System.out.println("Les dues cadenes són iguals");
}
```

Les dues cadenes són iguals

18.4.5. *length(): int*

Retorna el nombre de caràcters de la cadena.

```
String s = "cadena";
System.out.printf("//%"s// té %d caràcters", s, s.length());
```

"cadena" té 6 caràcters

18.4.6. *charAt(index: int): char*

Retorna el caràcter de la cadena de la posició especificada per **index**.

```
String s = "calamar";
for (int i = 0; i < s.length(); i++) {
    System.out.println(s.charAt(i));
}
```

c
a
l
a
m
a
r

18.4.7. *substring(beginIndex: int): String*

Retorna una subcadena de la cadena que comença al caràcter **beginIndex** i s'estén fins al final de la cadena.

```
String s = "En Pau s'ha fet mal.";
System.out.println(s.substring(7));
```

s'ha fet mal.

18.4.8. *substring(beginIndex: int, endIndex: int): String*

Retorna una subcadena de la cadena que comença al caràcter **beginIndex** i s'estén fins a **endIndex - 1**.

```
String s = "En Pau s'ha fet mal.";
System.out.println(s.substring(3,6));
```

```
Pau
```

18.4.9. *toLowerCase(): String*

Retorna una nova cadena amb **tots** els caràcters en minúscula.

```
String s = "AbCdE";
System.out.println(s.toLowerCase());
```

```
abcde
```

18.4.10. *toUpperCase(): String*

Retorna una nova cadena amb **tots** els caràcters en majúscula.

```
String s = "AbCdE";
System.out.println(s.toUpperCase());
```

```
ABCDE
```

18.4.11. *trim(): String*

Retorna una nova cadena sense els espais en blanc de la dreta i de l'esquerra.

```
String s = "    hola    ";
System.out.println(s.length());
System.out.println(s.trim().length());
```

```
10  
4
```

18.4.12. *replace(oldChar: char, newChar: char): String*

Retorna una nova cadena que reemplaça totes les aparicions del **caràcter oldChar** per **newChar**.

```
String s = "abracadabra";  
System.out.println(s.replace('a', 'u'));
```

```
ubrucudubru
```

18.4.13. *replaceFirst(oldString: String, newString: String): String*

Retorna una nova cadena que reemplaça la primera aparició de la cadena **oldString** per la cadena **newString**.

```
String s = "Plou poc, però per poc que plou, plou prou.";  
s.replaceFirst("poc", "molt");  
System.out.println(s.replaceFirst("poc", "molt"));
```

```
Plou molt, però per poc que plou, plou prou.
```

18.4.14. *replaceAll(oldString: String, newString: String): String*

Retorna una nova cadena que reemplaça totes les aparicions de la cadena **oldString** per la cadena **newString**.

```
String s = "Plou poc, però per poc que plou, plou prou.";  
System.out.println(s.replaceAll("o", ""));
```

```
Plu pc, però per pc que plu, plu pru.
```

18.4.15. *split(delimiter: String): String[]*

Retorna una matriu de cadenes consistent en totes les subcadenes delimitades per **delimiter**.

```
String s = "Plou poc, però per poc que plou, plou prou.";
String[] paraules = s.split("poc");
for (int i = 0; i < paraules.length; i++) {
    System.out.println(paraules[i]);
}
```

```
Plou
, però per
que plou, plou prou.
```

18.4.16. *indexOf(ch: char): int*

Retorna la primera ocurrència de ch dins de la cadena. Retorna -1 si no es troba el caràcter.

```
String s = "abcdeabcdeabcde";
System.out.println(s.indexOf('d'));
```

```
3
```

18.4.17. *indexOf(ch: char, fromIndex: int): int*

Retorna la primera ocurrència de ch després de **fromIndex** dins de la cadena. Retorna -1 si no es troba el caràcter.

18.4.18. *indexOf(s: String): int*

Retorna la primera ocurrència de s dins de la cadena. Retorna -1 si no es troba la subcadena.

18.4.19. *indexOf(s: String, fromIndex: int): int*

Retorna la primera ocurrència de s dins de la cadena després de **fromIndex**. Retorna -1 si no es troba la subcadena.

18.4.20. *lastIndexOf(ch: int): int*

Retorna la última ocurrència de **ch** dins de la cadena. Retorna -1 si no es troba el caràcter.

```
String s = "abcdeabcdeabcde";
System.out.println(s.lastIndexOf('d'));
```

```
13
```

18.4.21. *lastIndexOf(ch: int, fromIndex: int): int*

Retorna la última ocurrència de **ch** abans de **fromIndex** dins de la cadena. Retorna -1 si no es troba el caràcter.

18.4.22. *lastIndexOf(s: String): int*

Retorna la última ocurrència de **s** dins de la cadena. Retorna -1 si no es troba la subcadena.

18.4.23. *lastIndexOf(s: String, fromIndex: int): int*

Retorna la última ocurrència de **s** dins de la cadena abans de **fromIndex**. Retorna -1 si no es troba la subcadena.

18.4.24. *format(format, item1, item2, ..., itemk): String*

Aquest mètode és molt similar al mètode **printf** i la cadena de format funciona de la mateixa manera.

```
String s = String.format("One: %d Two: %d Three: %d", 10, 20, 30);
System.out.println(s);
```

```
One: 10 Two: 20 Three: 30
```

18.5. Conversió entre cadenes i altres tipus de dades

Les cadenes **no** són matrius però es pot convertir les unes en les altres.

18.5.1. **toCharArray(): char[]**

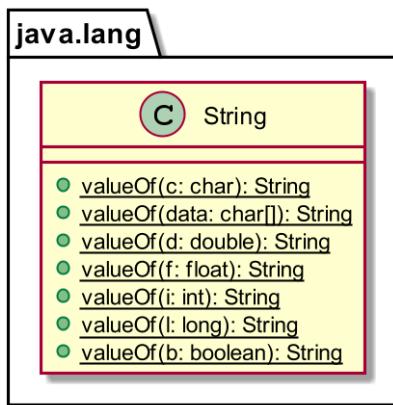
Converteix la cadena en una matriu de caràcters.

```
String s = "abcdeabcdeabcde";
char[] characters = s.toCharArray();
for(int i = 0; i < characters.length-1; i++) {
    System.out.print(characters[i] + ", ");
}
System.out.print(characters[characters.length-1]);
```

```
a, b, c, d, e, a, b, c, d, e, a, b, c, d, e
```

18.5.2. **valueOf(...): String**

La família de mètodes **valueOf** converteixen tipus de dades primitius en Strings.



Per exemple:

```
char[] c = {'h', 'o', 'l', 'a'};
System.out.println(String.valueOf(c));
```

```
hola
```



Cridar un mètode directament des d'un literal de cadena és vàlid.

```
System.out.println("Hola".charAt(0));
```

```
H
```

18.6. Classes embolcall o Wrapper Classes

Una característica important dels tipus per referència és que les funcionalitats que gestionen el tipus en qüestió es troben d'alguna manera dins del tipus i s'accedeixen a elles a través de l'operador `(.)`.

Amb els tipus primitius **això no és possible**, com que només s'emmagatzema el valor que contenen no hi ha possibilitat d'emmagatzemar altres coses i per tant els tipus primitius no tenen funcionalitats associades.

A vegades però fan falta, suposem que estem treballant amb el tipus primitiu **char**.

Tenim `char a = 'a'`; i volem passar la variable **a** a majúscula.

Seria lòtic poder fer una cosa semblant al que es fa amb les cadenes i fer `a.toUpperCase()`. Però això no és possible ja que **char** és un tipus primitiu i no té funcionalitats associades.

Tenim dues opcions:

1. Convertim el **char** a **String** i treballem amb les funcionalitats de **String**.

```
char a = 'a';
String s = String.valueOf(a);
System.out.println(s.toUpperCase());
```

2. o bé treballem amb una classe embolcall.

```
char a = 'a';
System.out.println(Character.toUpperCase(a));
```

Tots els tipus primitius tenen un tipus per referència associat. Per exemple, si ens cal un enter podem crear un enter de tipus primitiu **int** o bé un enter de tipus per referència **Integer**.

Treballar amb tipus primitius és molt més **ràpid**, treballar amb tipus per referència dona accés a les **funcionalitats associades al tipus**.

Taula 18.1. Relació entre tipus primitius i tipus per referència

Tipus primitiu	Tipus per referència
char	Character
boolean	Boolean
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

18.7. Conversió entre cadenes i números

Les classes embolcall numèriques proporcionen mètodes per convertir una cadena en una dada numèrica.

- Per convertir un **String** a un **int** cal utilitzar el mètode **Integer.parseInt(intString)**.
- Per convertir un **String** a un **Double** cal utilitzar el mètode **Double.parseDouble(doubleString)**.
- Per convertir una **número** a un **String** es pot utilitzar el mètode **String.valueOf(número)**.

```
String intString = "23";
int valor = Integer.parseInt(intString);
```

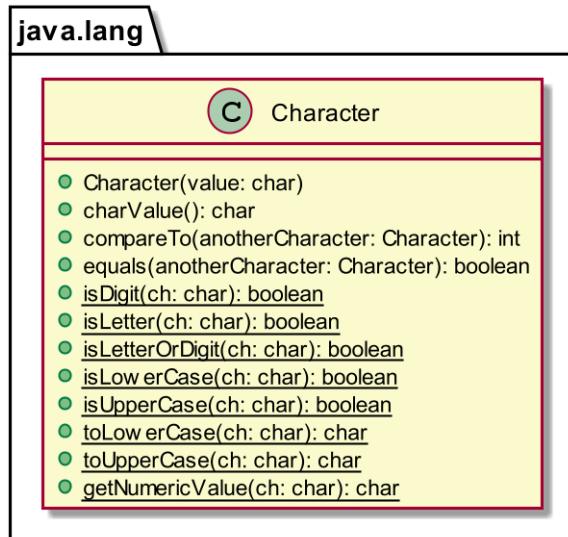
18.8. El tipus de dades Character

Per crear un objecte de tipus Character:

```
Character c;
```

```
c = new Character('d');
```

18.9. Mètodes de la classe Character



charValue(): char

Retorna el valor de char per aquest objecte.

compareTo(anotherCharacter: Character): int

Compara aquest Character amb un altre.

equals(anotherCharacter: Character): boolean

Retorna true si aquest Character és igual a un altre.

isDigit(ch: char): boolean

Retorna true si el char especificat és un dígit.

isLetter(ch: char): boolean

Retorna true si el char especificat és una lletra.

isLetterOrDigit(ch: char): boolean

Retorna true si el char especificat és una lletra o un dígit.

isLowerCase(ch: char): boolean

Retorna true si el char especificat és una lletra minúscula.

isUpperCase(ch: char): boolean

Retorna true si el char especificat és una lletra majúscula.

toLowerCase(ch: char): char

Retorna la lletra minúscula del char especificat.

toUpperCase(ch: char): char

Retorna la lletra majúscula del char especificat.

getNumericValue(ch: char)

Retorna el valor numèric especificat pel caràcter Unicode que representa.

Si el caràcter no representa un valor numèric el mètode retorna un **-1**. Si el caràcter té un valor numèric que no es pot representar com un enter no negatiu, per exemple una fracció, el mètode retorna un **-2**.

18.10. Lectura de cadenes amb la classe Scanner

Fins ara quan creàvem un objecte de la classe **Scanner** ho fèiem passant-li a l'objecte una referència a la entrada de dades estàndard, **System.in**, d'aquesta manera l'objecte **Scanner** obtenia el flux de dades amb el que treballar d'aquesta entrada de dades.

De forma similar, al crear un objecte de la classe **Scanner** se li pot passar una cadena enlloc de **System.in**, a partir d'aquest moment l'objecte **Scanner** tractara la cadena passada com un flux de dades d'entrada.

Vegem-ho amb un exemple:

```
import java.util.Scanner;

public class TestScanner {

    public static void main(String[] args) {
        String s = "test 10 3,2";
        Scanner sc = new Scanner(s);

        System.out.println(sc.next());
        System.out.println(sc.nextInt());
        System.out.println(sc.nextDouble());
    }
}
```

```
}
```

```
test
10
3.2
```

18.11. Cadenes i la configuració regional

El tractament i la comparació de cadenes en idiomes que utilitzen conjunts de caràcters diferents dels americans és delicada i complexa.

Dues classes que poden ser útils són la classe **Locale** i la classe **Collator**. La primera encapsula les característiques d'una configuració regional i la segona permet fer comparacions en base a un **Locale** determinat.

Un exemple d'ús:

```
public class ComparacioCadenes {

    public static void main(String[] args) {

        Locale caLocale = new Locale("ca", "ES");

        Collator c = Collator.getInstance(caLocale);
        c.setStrength(Collator.PRIMARY);

        String str1 = "a";
        String str2 = "à";
        boolean diff = c.equals(str1, str2);
        System.out.print("Comparem segons la classe String: ");
        System.out.println("Són iguals? : " + diff);

        System.out.print("Comparem segons la classe Collator: ");
        diff = str1.equals(str2);
        System.out.println("Són iguals? : " + diff);
    }
}
```

```
Comparem segons la classe String: Són iguals? : true
Comparem segons la classe Collator: Són iguals? : false
```

18.12. Classes StringBuilder i StringBuffer

Les classes **StringBuilder** i **StringBuffer** són similars a la classe **String** excepte que **String** és immutable i **StringBuilder** i **StringBuffer** no ho són.

En general les classes **StringBuilder** i **StringBuffer** es poden utilitzar a qualsevol lloc on es pugui utilitzar un **String**. Aquestes dues classes són més flexibles que **String** ja que permeten **afegir** o **inserir** nou contingut a una cadena.

La classe **StringBuilder** és molt similar a **StringBuffer** amb la diferència que els mètodes que permeten modificar el "buffer" de dades intern a **StringBuffer** estan sincronitzats, és a dir, *una sola "tasca" pot executar aquests mètodes.

 StringBuilder	
<ul style="list-style-type: none"> ● <code>StringBuilder()</code> ● <code>StringBuilder(capacity: int)</code> ● <code>StringBuilder(s: String)</code> ● <code>append(data: char[]): StringBuilder</code> ● <code>append(data: char[], offset: int, len: int): StringBuilder</code> ● <code>append(v: aPrimitiveType): StringBuilder</code> ● <code>append(s: String): StringBuilder</code> ● <code>delete(startIndex: int, endIndex: int): StringBuilder</code> ● <code>deleteCharAt(index: int): StringBuilder</code> ● <code>insert(index: int, data: char[], offset: int, len: int): StringBuilder</code> ● <code>insert(offset: int, data: char[]): StringBuilder</code> ● <code>insert(offset: int, b: aPrimitiveType): StringBuilder</code> ● <code>insert(offset: int, s: String): StringBuilder</code> ● <code>replace(startIndex: int, endIndex: int, s: String): StringBuilder</code> ● <code>reverse(): StringBuilder</code> ● <code>setCharAt(index: int, ch: char): void</code> ● <code>toString(): String</code> 	

Comparem el temps d'execució dels dos programes següents, fan el mateix, un directament concatenant cadenes i l'altre utilitzant un **StringBuilder**.

Utilitzant Strings.

```
public class Cadenes {

    public static void main(String[] args) {
        String s = "";
```

```
    for (int i = 0; i < 100000; i++) {
        s += ", " + i;
    }
    System.out.println(s.length());
}
}
```

```
688890
BUILD SUCCESSFUL (total time: 33 seconds)
```

Utilitzant StringBuffer.

```
public class Cadenes {

    public static void main(String[] args) {
        String s = "";
        StringBuilder sb = new StringBuilder();

        for (int i = 0; i < 100000; i++) {
            sb.append(", " + i);
        }
        s = sb.toString();
        System.out.println(s.length());
    }
}
```

```
688890
BUILD SUCCESSFUL (total time: 0 seconds)
```

19

Mètodes (funcions)

Definir una **funció** consisteix en essència en posar un nom a un bloc de codi, de manera que fent una crida al nom s'executa el codi associat.

A més permeten la entrada i el retorn de valors variables.

Mètodes o Funcions

En aquesta UF considerarem els termes **mètode** i **funció** com a termes equivalents i intercanviables.

En realitat els conceptes són similars però s'utilitza el terme **funció** en els llenguatges que segueixen el paradigma de **programació procedimental** i el terme **mètode** en els llenguatges basats en la **programació orientada a objectes**.

Java és per construcció un llenguatge de programació orientat a objectes, per tant, tècnicament, hauríem de parlar de mètodes enllloc de funcions.

L'ús dels mètodes proporciona una sèrie d'avantatges importants:

Augmenta la claredat del codi font

El flux principal de l'aplicació serà essencialment una seqüència de crides a funcions, els detalls més tècnics no seran visibles al flux principal ja que estaran encapsulats dins de les pròpies funcions.

Facilita les proves

Les funcions es poden provar de forma independent i integrar posteriorment al programa.

Facilita el manteniment

Permeten eliminar el codi duplicat i per tant el manteniment s'ha de realitzar en un únic lloc.

Facilita la reutilització del codi

Una funció es pot utilitzar a més d'un programa sense necessitat de fer-ne una còpia directament al codi font del programa.

Gestió d'errors independent

Permet que cada mètode gestioni els propis errors de manera interna. Això no sempre és necessari o desitjable però ajuda a tractar els mètodes com a caixes negres.



El terme **funció** està molt associat a llenguatges procedurals, en la terminologia dels llenguatges orientats a objectes reben el nom de **mètodes**

19.1. Definir un mètode

La sintaxis per a la definició d'un mètode és:

```
modificador tipusValorRetorn nomMètode(llista de paràmetres) {  
}
```

modificador

De moment tots els mètodes que declarem tindran el modificador de **public static**. La resta de modificadors apareixeran quan treballarem la programació orientada a objectes.

tipusValorRetorn

Un mètode admet uns paràmetres, fa un processament de dades i retorna un resultat, el tipus de valor de retorn és el nom del tipus de dades que retornarà el mètode.

Un tipus de retorn especial és la paraula clau **void** que indica que el mètode no retorna cap valor.

nomMètode

El nom del mètode és el nom pel qual és cridarà el mètode per executar-lo. Pot ser qualsevol nom admissible en Java però per conveni estarà en notació **camelCase** i començarà en **minúscula**.

llista de paràmetres

És una llista separada per comes de declaracions de variables que representen els paràmetres d'entrada del mètode.

Exemple:

```
public static void hola() {  
    System.out.println("Hola");  
}
```



El nom del mètode junt amb el **tipus** tots els paràmetres d'entrada **ordenats** s'anomena la **signatura del mètode**. El tipus del valor de retorn **no** forma part de la signatura del mètode.

19.2. Paraula clau return

Quan es crida un mètode aquest s'executa fins que arriba al final o bé fins que es troba la paraula clau **return**.

return s'utilitza de dues maneres:

1. Per finalitzar l'execució del mètode.
2. Per finalitzar l'execució del mètode i indicar quin és el seu valor de retorn.

Exemples:

Exemple 1.

```
public static void hola() {  
    System.out.println("Hola");  
    return; // Les següents línies del mètode no s'executaran mai  
    System.out.println("Adeu");  
}
```

Exemple 2.

```
public static int suma(int a, int b) {
    return a+b;
}
```

19.3. Cridar un mètode

Cridar un mètode executa el codi dins del mètode, si un mètode no es crida, no s'executa el seu codi.

Existeixen dues maneres de cridar un mètode en funció de si aquest retorna un valor o no:

- Si el mètode no retorna un valor n'hi ha prou en posar el seu nom i entre parèntesis tots els valors d'entrada demanats.

```
public class Hola {
    public static void main(String args[]) {
        hola();
    }

    public static void hola() {
        System.out.println("Hola");
    }
}
```

- Si el mètode retorna un valor, o s'emmagatzema en una variable o es passa com a valor a un altre mètode.

```
public class FuncionsMatematiques {
    public static void main(String args[]) {
        int major = max(12, 56); // Posem el resultat en una
        variable

        System.out.println(max(12,34)); // o passem directament el
        valor de retorn com a paràmetre d'entrada d'un altre mètode
    }

    public static int max(int a, int b) {
        if (a > b) {
            return a;
        } else {
```

```

        return b;
    }
}

```

19.4. Ubicació dels mètodes

Els mètodes en Java es troben sempre dins d'una estructura **class**, un a continuació de l'altre.

- Si es comença la definició d'un mètode abans de tancar l'anterior es produirà un error de sintaxis.
- Si es defineix un mètode fora d'una estructura class es produirà un error.

Exemples:

Exemple 1.

```

public class Hola {
    public static void main(String args[]) {
        public static void hola() {
            System.out.println("Hola");
        }
    }
}

```

L'Exemple 1 genera un error, els mètodes no poden estar dins d'un altre mètode, han d'estar a un nivell de la classe.

Exemple 2.

```

public class Hola {
    public static void main(String args[]) {

    }
} // tanquem la classe
public static void hola() {
    System.out.println("Hola");
}

```

L'Exemple 2 generarà un error, els mètodes han d'estar dins d'una estructura class.

19.5. La pila de crides a funcions

Cada cop que s'invoca un mètode, el sistema crea un **registre d'activació**, "stack frame" en anglès, que emmagatzema els paràmetres i les variables pel mètode. Aquest registre s'emmagatzema a una àrea de memòria anomenada **pila de crides a funcions** (call stack), o simplement **la pila**.

Quan un mètode crida a un altre mètode, el registre d'activació del mètode inicial es manté intacte i un nou registre d'activació es crea a la pila pel nou mètode cridat.

Quan un mètode acaba i retorna a qui l'ha cridat, el registre d'activació del mètode finalitzat s'elimina de la pila, i per tant totes les variables locals de tipus primitius totes les referències a objectes s'eliminen.



Que s'eliminint de la pila les referències als objectes no implica que s'eliminint els propis objectes, ja que aquests que viuen al "heap" i la sortida d'un mètode només afecta al "stack".

La pila emmagatzema els registres d'activació en una estructura **lifo**, "last-in, first-out", l'últim registre d'activació creat és el primer que s'eliminarà.

Per exemple:

```
public class TestMax {
    public static void main(String[] args) {
        int i = 5;
        int j = 2;
        int k = max(i, j);

        System.out.println("El màxim entre " + i + " i " + j + " és " +
                           k);
    }

    public static int max(int num1, int num2) {
        int resultat;

        if (num1 > num2)
            resultat = num1;
        else
```

```

        resultat = num2;

        return resultat;
    }
}

```

El màxim entre 5 i 2 és 5

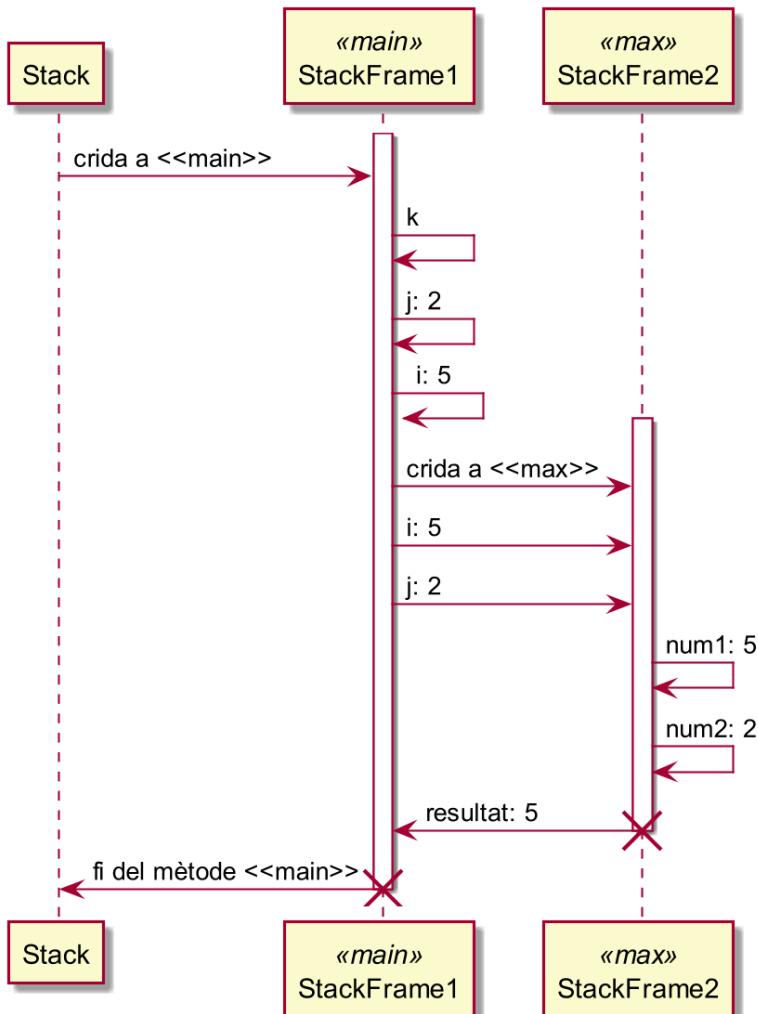


Figura 19.1. Quan s'invoca el mètode max, el flux de control es transfereix al mètode max. Quan el mètode max finalitza, el control retorna al mètode main.

19.6. Passar arguments per valor

Els arguments en Java es passen per valor.

Quan s'invoca a un mètode amb un argument, el que rep el mètode és una copia del valor de l'argument, no l'argument en sí, això es coneix com a **pas per valor**.

Això implica que si el mètode canvia el valor de l'argument no està afectant al contingut de les variables del bloc que ha cridat el mètode.

Per exemple:

```
public class TestPasPerValor {
    public static void main(String[] args) {
        int valorMain = 10;

        System.out.println("Valor main inicial : " + valorMain);
        testValor(valorMain);
        System.out.println("Valor main final : " + valorMain);

    }

    public static void testValor(int valor) {
        valor = valor + 1;
        System.out.println("Valor testValor final : " + valor);
    }
}
```

```
Valor main inicial: 10
Valor testValor final: 11
Valor main final: 10
```

Aquest comportament pot portar a codificacions defectuoses. El següent exemple intenta codificar una funció que intercanvii dos valors.

El contingut de n1 i n2 és una copia dels valors passats i per tant modificar aquestes variables no modifica les variables originals.

```
public class TestPassByValue {
    public static void main(String[] args) {
        int num1 = 1;
```

```
int num2 = 2;

System.out.println("Abans d'invocar el mètode swap, num1 és " +
num1 + " i num2 és " + num2);

// Intentem intercanviar els valors de num1 i num2
swap(num1, num2);

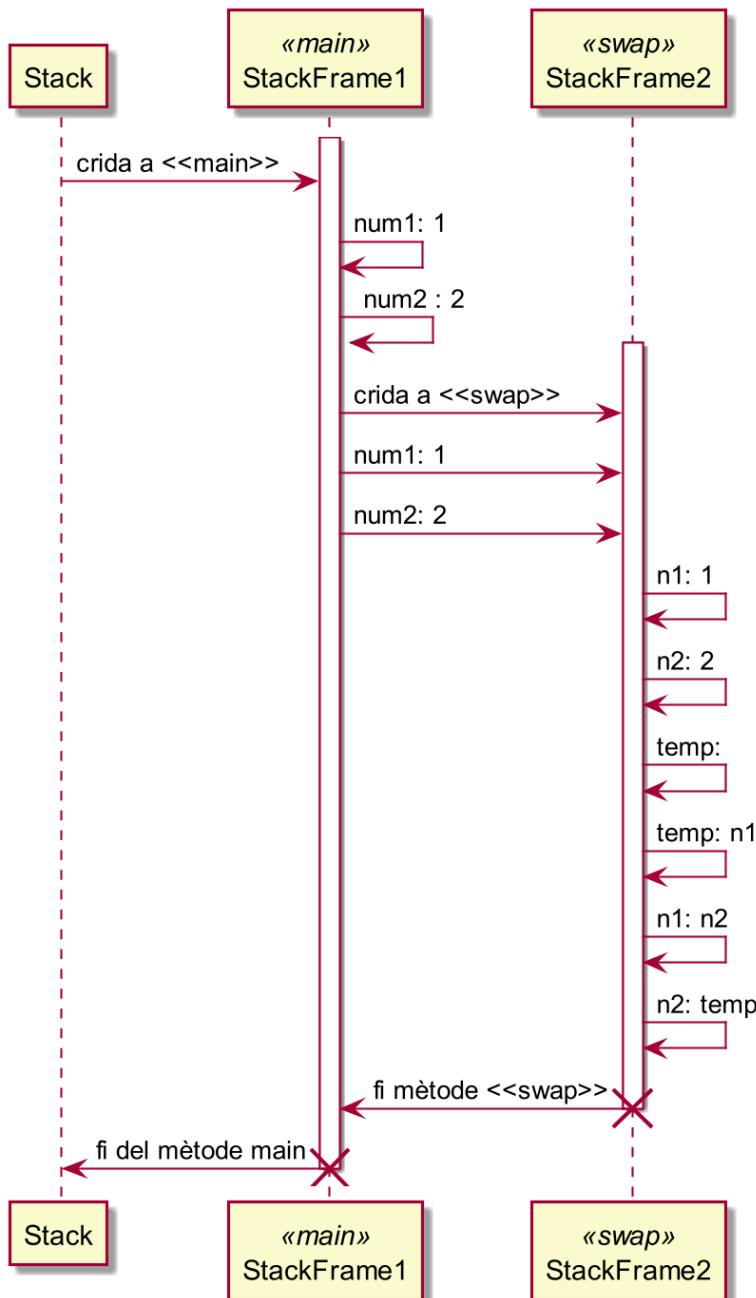
System.out.println("Després d'invocar el mètode swap, num1 és " +
+ num1 + " i num2 és " + num2);
}

public static void swap(int n1, int n2) {
    System.out.println("\tDins del mètode swap");
    System.out.println("\t\tAbans de l'intercanvi, n1 és " + n1 + " i n2 és " + n2);

    // Intercanviem els valors
    int temp = n1;
    n1 = n2;
    n2 = temp;

    System.out.println("\t\tDesprés de l'intercanvi, n1 és " + n1 +
+ " i n2 és " + n2);
}
}
```

```
Abans d'invocar el mètode swap, num1 és 1 i num2 és 2
    Dins del mètode swap
        Abans de l'intercanvi, n1 és 1 i n2 és 2
        Després de l'intercanvi, n1 és 2 i n2 és 1
Després d'invocar el mètode swap, num1 és 1 i num2 és 2
```



De forma similar, si enlloc de treballar amb variables de tipus **int**, primitives i per tant amb el seu contingut allotjat al "stack", treballem amb variables de tipus

Integer, objectes allotjats al "heap" però amb una referència emmagatzemada al "stack", tindrem un comportament similar al anterior.

El contingut de n1 i n2 és una copia de la referència als objectes emmagatzemats al "heap" per tant intercanviar les referències entre n1 i n2 no intercanvia el contingut de les variables originals.

```
public class TestPassByValue2 {

    public static void main(String[] args) {
        Integer num1 = new Integer(1);
        Integer num2 = new Integer(2);

        System.out.println("Abans d'invocar el mètode swap, num1 és " +
                           num1 + " i num2 és " + num2);

        // Intentem intercanviar els valors de num1 i num2
        swap(num1, num2);

        System.out.println("Després d'invocar el mètode swap, num1 és " +
                           num1 + " i num2 és " + num2);
    }

    public static void swap(Integer n1, Integer n2) {
        System.out.println("\tDins del mètode swap");
        System.out.println("\t\tAbans de l'intercanvi, n1 és " + n1 + " i n2 és " + n2);

        // Intercanviem els valors
        int temp = n1;
        n1 = n2;
        n2 = temp;

        System.out.println("\t\tDesprés de l'intercanvi, n1 és " + n1 +
                           " i n2 és " + n2);
    }
}
```

```
Abans d'invocar el mètode swap, num1 és 1 i num2 és 2
    Dins del mètode swap
        Abans de l'intercanvi, n1 és 1 i n2 és 2
        Després de l'intercanvi, n1 és 2 i n2 és 1
Després d'invocar el mètode swap, num1 és 1 i num2 és 2
```

19.7. Referències passades per valor

Encara que els objectes es passen per valor i els mètodes no poden modificar les referències, els valors de les adreces de memòria que apunten a l'objecte, els mètodes **si que poden modificar l'estat d'un objecte**. Vegem-ho:

```
class Contenidor {

    int enter;
}

public class TestPassByValue3 {

    public static void main(String[] args) {
        Contenidor num1 = new Contenidor();
        Contenidor num2 = new Contenidor();
        num1.enter = 1;
        num2.enter = 2;

        System.out.println("Abans d'invocar el mètode swap, num1 és " +
                           num1.enter + " i num2 és " + num2.enter);

        // Intentem intercanviar els valors de num1 i num2
        swap(num1, num2); ①

        System.out.println("Després d'invocar el mètode swap, num1 és " +
                           num1.enter + " i num2 és " + num2.enter); ②
    }

    public static void swap(Contenidor n1, Contenidor n2) {
        System.out.println("\tDins del mètode swap"); ③
        System.out.println("\t\tAbans de l'intercanvi, n1 és " +
                           n1.enter + " i n2 és " + n2.enter);

        // Intercanviem els valors
        int temp = n1.enter; ④
        n1.enter = n2.enter;
        n2.enter = temp;

        System.out.println("\t\tDesprés de l'intercanvi, n1 és " +
                           n1.enter + " i n2 és " + n2.enter);
    }
}
```

```
}
```

Abans d'invocar el mètode swap, num1 és 1 i num2 és 2

Dins del mètode swap

Abans de l'intercanvi, n1 és 1 i n2 és 2

Després de l'intercanvi, n1 és 2 i n2 és 1

Després d'invocar el mètode swap, num1 és 2 i num2 és 1

- ① **num1** i **num2** apunten a dos objectes al "heap" que contenen el valor 1 i 2 respectivament.
- ③ Les variables **n1** i **n2** són còpies de les variables **num1** i **num2**, per tant **n1** i **num1** estan apuntant a la mateixa estructura de dades al "heap", la que manté el valor de 1, de forma similar, **n2** i **num2** apunten a la mateixa estructura en el "heal", la que manté el valor de 2.
- ④ Intercanviem el **contingut** de les estructures de dades mantenint les referències a aquetses estructures intactes.
- ② Les adreces de memòria a les que apunten **num1** i **num2** no han estat modificades però **si que ho ha estat el seu contingut** que s'ha intercanviat fent l'efecte que les dues variables han intercanviat els valors.

19.8. Passar matrius als mètodes

De forma similar a l'exemple anterior, quan es passa una matriu a un mètode està passant una referència a l'objecte. De la mateixa manera que es poden passar tipus primitius com a paràmetres dels mètodes també es poden passar tipus per referència, o en particular arrays.

Per exemple, el següent mètode mostra els elements d'una matriu d'enters.

```
public static void mostrarPatriu(int[] matriu) {
    for (int i = 0; i < matriu.length; i++) {
        System.out.print(matriu[i] + " ");
    }
}
```

Per invocar-lo només cal passar-li una matriu, per exemple:

```
mostrarMatriu(new int[]{1, 2, 5, 8, 9});
```



Com ja s'ha dit, Java utilitza el pas per valor per passar arguments als mètodes, en el cas de les matrius també és així. No obstant hi ha diferències importants entre passar arguments de tipus per referència i arguments de tipus per valor.

Vegem-ho:

El valor d'un argument matriu és una referència a la matriu, quan es passa aquesta referència al mètode es fa una copia per valor de la referència. En aquest moment tenim dues referències diferents que tenen el mateix valor, apunten a l'objecte matriu, per tant qualsevol canvi a la matriu des d'una de les referències es podrà veure des de l'altra ja que en realitat l'objecte és el mateix.

Per exemple:

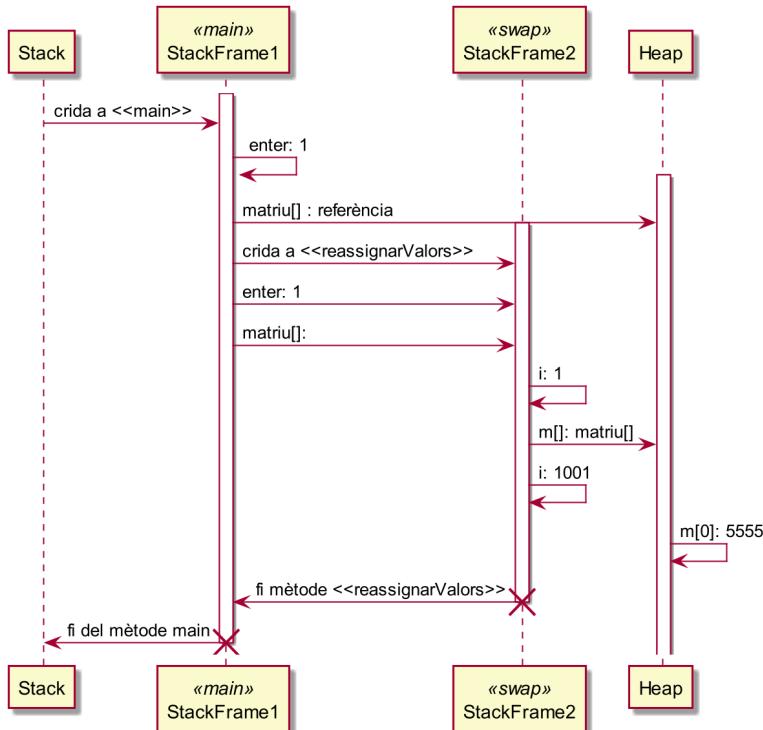
```
public class Test {
    public static void main(String[] args) {
        int enter = 1;
        int[] matriu = new int[10];
        matriu[0] = 1;

        reassignarValors(enter, matriu);

        System.out.println("enter és " + enter);
        System.out.println("matriu[0] és " + matriu[0]);
    }

    public static void reassignarValors(int i, int[] m) {
        i = 1001;
        m[0] = 5555;
    }
}
```

```
x és 1
y[0] és 5555
```



El següent exemple mostra dos mètodes el primer no es correcte ja que treballa amb tipus primitius, el segon és correcte ja que treballa amb tipus per referència.

```
public static void swap(int n1, int n2) {
    int temp = n1;
    n1 = n2;
    n2 = temp;
}
```

```
public static void swapFirstTwoInArray(int[] array) {
    int temp = array[0];
    array[0] = array[1];
    array[1] = temp;
}
```

El contingut de array és una copia de la referència a nums, per tant nums i array fan referència al a la mateixa matriu. Com a conseqüència canviar el contingut de la matriu des d'una referència o des de l'altre és indiferent.

```

public class TestPassByValue3 {

    public static void main(String[] args) {
        int[] nums = {1, 2};
        System.out.println("Abans d'invocar el mètode swap, nums[0] és "
+ nums[0] + " i nums[1] és " + nums[1]);

        // Intentem intercanviar els valors de num1 i num2
        swapFirstTwoInArray(nums);

        System.out.println("Després d'invocar el mètode swap, nums[0] és "
" + nums[0] + " i nums[1] és " + nums[1]);
    }

    public static void swapFirstTwoInArray(int[] array) {
        System.out.println("\tDins del mètode swap");
        System.out.println("\t\tAbans de l'intercanvi, array[0] és " +
array[0] + " i array[1] és " + array[1]);

        // Intercanviem els valors
        int temp = array[0];
        array[0] = array[1];
        array[1] = temp;

        System.out.println("\t\tDesprés de l'intercanvi, array[0] és " +
array[0] + " i array[1] és " + array[1]);
    }
}

```

Abans d'invocar el mètode swap, nums[0] és 1 i nums[1] és 2
Dins del mètode swap
Abans de l'intercanvi, array[0] és 1 i array[1] és 2
Després de l'intercanvi, array[0] és 2 i array[1] és 1
Després d'invocar el mètode swap, nums[0] és 2 i nums[1] és 1



Treballar amb paràmetres per referència proporciona una manera de retornar més d'un valor des d'un mètode.

No obstant no s'hauria d'utilitzar aquesta possibilitat ja que no hi ha manera de saber que un paràmetre es de sortida, enllot que d'entrada mirant la crida d'una funció cosa que complica la comprensió del codi.

Quan treballem amb objectes els paràmetres de sortida seran innecessaris.

19.9. Tornar una matriu des d'un mètode

De la mateixa manera que un mètode pot rebre u na matriu també pot retornar-la.

Per exemple, el següent mètode retorna una matriu amb els elements invertits d'una altra matriu.

```
public static int[] invertirMatriu(int[] list) {  
    int[] result = new int[list.length];  
  
    for (int i = 0, j = result.length - 1; i < list.length; i++, j--) {  
        result[j] = list[i];  
    }  
  
    return result;  
}
```

19.10. Arguments per la línia de comandes

El mètode main per rebre arguments de tipus cadena des de la línia de comandes. De fet la declaració del mètode main indica que admet una matriu de cadenes.

```
public static void main(String[] args)
```

Funciona com qualsevol altre mètode amb un paràmetre d'entrada de tipus matriu amb la peculiaritat que se li poden passar paràmetres des de la línia de comandes al executar el programa.

El següent exemple passa tres cadenes arg0, arg1 i arg2 al programa TestMain:

```
public class TestMain {  
    public static void main(String[] args) {  
        for (int i = 0; i < args.length; i++) {  
            System.out.println(args[i]);  
        }  
    }  
}
```

```
}
```

```
java TestMain arg0 arg1 arg2
```

Les cadenes no han d'anar entre cometes amb l'excepció que la cadena passada tingui espais, per exemple:

```
java TestMain "Això és una prova" 1 2 3
```

A l'invocar el mètode main l'interpret de Java crea una matriu per emmagatzemar tots els valors passats.



En cas de no passar cap argument es crea una matriu String[0] amb zero elements.

19.11. Sobrecàrrega de mètodes

La sobrecàrrega de mètodes permet definir diferents mètodes amb el mateix nom sempre i quant les **signatures** dels mètodes siguin diferents.

Per exemple, els següents mètodes són correctes en Java i poden estar definits dins de la mateixa classe,

```
// Versió 1
public static int max(int n1, int n2) {
    return n1 >= n2 ? n1 : n2;
}

// Versió 2
public static double max(double n1, double n2) {
    return n1 >= n2 ? n1 : n2;
}

// Versió 3
public static double max(double n1, double n2, double n3) {
    return max(max(n1, n2), n3);
}
```

A l' hora d'invocar un dels mètodes sobrecarregats Java tindrà en compte la quantitat i tipus dels paràmetres de la crida per determinar quina de les versions sobracarregades es crida.

Per exemple:

```
max(3, 5); // Cridarà la versió 1 del mètode
max(2.3, 5.6); // Cridarà la versió 2 del mètode
max(3, 4.5); // Cridarà la versió 2 del mètode
```



La sobrecàrrega de mètodes pot ajudar a fer els programes més clars i més llegibles.

La mateixa funció amb diferents paràmetres hauria de tenir el mateix nom.



Els mètodes sobrecarragats han de tenir diferent llista de paràmetres d'entrada.

No es poden sobrecarregar mètodes basant-se en diferents modificadors o en el tipus de valor de retorn.



És possible que en determinades ocasions el compilador sigui incapaç de decidir quina de les versions d'un mateix mètode ha de cridar.

Això es coneix com una invocació ambigua i genera un error de compilació. Per exemple:

```
public class SobrecarregAmbigua {
    public static void main(String[] args) {
        System.out.println(max(1, 2));
    }
    public static double max(int n1, double n2) {
        if (num1 > num2)
            return num1;
        else
            return num2;
    }
    public static double max(double n1, int n2) {
        if (num1 > num2)
```

```

        return num1;
    else
        return num2;
}
}

```

19.12. Àmbit de les variables

L'àmbit d'una variable fa referència a la part del programa on es pot referenciar aquesta variable.

19.12.1. Variables d'àmbit de classe

Les variables es poden declarar a nivell de classe, en aquest cas, com a les funcions, cal posar la paraula clau *static* abans del tipus de dades de la declaració.

Per exemple:

```

public class Test {
    public static int variableClasse;

    public static void main(String[] args) {
    }
}

```

Una variable declarada a nivell de classe és accessible des de qualsevol mètode de la classe.



No penseu en les variables de classe com a alternativa a les variables d'entrada dels mètodes.

Tot i que sembli més còmode treballar amb variables de classe que amb paràmetres d'entrada modificar variables de classe des dels mètodes pot generar efectes secundaris molt difícils de detectar a la resta del programa.

Exemple de mal ús de les variables de classe.

```

public class Test {
    public static int a, b, resultatSuma;
}

```

```

public static void main(String[] args) {
    a = 2;
    b = 4;
    suma();
    System.out.println("Resultat: " + resultatSuma);
}

public static void suma() {
    resultatSuma = a + b;
}
}

```



Les variables definides a nivell de classe poden tenir diferents àmbits d'accessibilitat.

Ho veurem en detall quan parlem de la programació orientada a objectes.

19.12.2. Constants d'àmbit de classe

De forma similar a la declaració de variables amb àmbit de classe també es poden declarar constants a nivell de classe. Per fer-ho cal afegir l'especificador *static* a la declaració d'ela constant.

```

public class Test {
    public static final int LONGITUD_MAXIMA = 10;

    public static void main(String[] args) {
    }
}

```

19.12.3. Variables d'àmbit de mètode

Una variable declarada dins d'un mètode és accessible únicament des de dins del mètode on s'ha declarat.

- Aquestes variables s'anomenen *variables locals*.
- Es creen al entrar al mètode i es destrueixen al sortir-ne.
- Més d'un mètode pot declarar la mateixa variable.
- Les variables locals s'han de declarar i assignar abans de poder ser utilitzades.

Per exemple:

```
public class Test {
    public static void main(String[] args) {
        int variableMain = 4;
        System.out.println(variableMetode); // Error variableMetode
només és accessible des de metode()
    }

    public void metode() {
        int variableMetode = 4;
        System.out.println(variableMain); // Error, variableMain no és
accessible des de metode
    }
}
```

19.12.4. Variables d'àmbit de bloc

Una variable declarada en un bloc es pot utilitzar únicament dins aquest bloc.

```
1 public static void metode() {
2     // Cos del mètode
3     for(int i = 0; i < 10; i++) {
4         // Bucle
5     }
6     System.out.println(i);
7 }
```

La línia 6 genera un error de compilació ja que la variable i només és accessible dins del bloc delimitat pel bucle for.



Les variables declarades dins d'un bloc només són accessibles des del bloc però es creen al entrar al mètode que les declara i es destrueixen al sortir-ne.

Per aquesta raó no és possible tenir una variable amb el mateix nom a nivell de mètode i a nivell de bloc (dins del mateix mètode), ja que la variable dins del bloc és en realitat la mateixa que la definida dins el mètode, el que passa és que fora del bloc no és accessible.

No obstant, si que és possible redclarar la variable en un altre bloc del mateix mètode.

```
1 public static void metode() {  
2     // Cos del mètode  
3     for(int i = 0; i < 10; i++) {  
4         // Bucle  
5     }  
6     if(true) {  
7         int i = 50;  
8         // codi  
9     }  
10    System.out.println(i);  
11 }
```

19.12.5. "shadowing"

Es possible definir una variable amb el mateix nom dins d'una classe i dins d'un dels seus mètodes. La raó d'això queda fora de l'abast d'aquesta uf.

En aquesta situació la variable a la que s'accedeix és a la versió de la variable la declaració de la qual queda "més a prop", és a dir, si la variable està declarada al mètode des d'on s'accedeix s'utilitzaria la versió definida al mètode i en cas contrari s'utilitzaria la versió definida a la classe.

Per exemple:

```
public class Variables {  
    int valor = 10;  
  
    public static void main (String [ ] args) {  
        int valor = 6;  
  
        System.out.println(valor); // valor = 6  
        metode1();  
        metode2();  
    }  
  
    public static void metode1() {  
        int valor = 7;  
        System.out.println(valor); // valor = 7
```

```
}

public static void metode2() {
    System.out.println(valor); // valor = 10
}
}
```

```
6
7
10
```

19.13. Llista d'arguments de longitud variable

Una construcció del llenguatge permet passar un nombre variable d'arguments del mateix tipus a un mètode i tractar-los com una matriu.

Per fer-ho cal declarar el paràmetre del mètode de la següent manera:

tipus... nomVariable

- Només pot haver-hi un paràmetre de longitud variable en la declaració d'un mètode i si hi és ha de ser l'últim paràmetre del mètode.
- Java tracta els paràmetres de longitud variable com a matrius.

```
public class VarArgsDemo {
    public static void main(String[] args) {
        printMax(34, 3, 3, 2, 56.5);
        printMax(new double[]{1, 2, 3});
    }

    public static void printMax(double... nombres) {
        if (nombres.length == 0) {
            System.out.println("No hi ha arguments");
            return;
        }

        double result = nombres[0];

        for (int i = 1; i < nombres.length; i++) {
            if (nombres[i] > result)
                result = nombres[i];
        }
    }
}
```

```
    }  
  
    System.out.println("El valor màxim és " + result);  
}  
}
```


20

Programació modular en Java

La **programació modular** és una tècnica de disseny de Software que emfatitza el separar la funcionalitat d'un programa en **mòduls** independents i intercanviables que continguin tot el necessari per executar un aspecte concret de la funcionalitat desitjada.

Si deixem de banda les característiques específiques de la programació orientada a objectes en Java, Java proporciona les següents construccions orientades a encapsular funcionalitat a diferents escales.

- Mètodes
- Classes
- Biblioteques
- Paquets .jar

20.1. Mètodes / funcions

Una funció és una seqüència d'instruccions que realitza una tasca específica tractada com a una unitat.

En funció dels diferents llenguatges de programació a vegades les funcions s'anomenen, **mètodes**, **routines**, **subrutines** o **subprogrames**.

Per exemple:

```
public static double suma(double... o) {  
    double suma = 0;  
    for (int i = 0; i < o.length; i++) {
```

```

        suma += o[i];
    }
    return suma;
}

```

20.2. Classes estàtiques

Una estructura **class** amb mètodes **static** representa un primer nivell d'agrupament de mètodes.

Vegem-ho amb un exemple:

Fitxer Operacions.java.

```

public class Operacions { ❶

    public static double producte(double... o) { ❷
        double producte = 1;
        for (int i = 0; i < o.length; i++) {
            producte *= o[i];
        }
        return producte;
    }

    public static double suma(double... o) { ❸
        double suma = 0;
        for (int i = 0; i < o.length; i++) {
            suma += o[i];
        }
        return suma;
    }
}

```

- ❶ Definim una estructura **class** de nom **Operacions** en un fitxer *.java* de nom **Operacions.java**.
- ❷ Afegim un mètode **static** dins de l'estructura **Operacions**.
- ❸ Afegim un segon mètode **static** dins de l'estructura **Operacions**.

donada l'estructura anterior, podem, des del mateix paquet cridar a qualsevol dels dos mètodes afegint el nom de la classe on es troben.

Per exemple,

```
public class Main {

    public static void main(String[] args) {
        System.out.println(Operaciones.producte(2, 3, 10));
        System.out.println(Operaciones.suma(1, 2, 3, 4, 5));
    }

}
```

60.0
15.0

20.3. Biblioteques

Les classes definides a l'apartat anterior es poden organitzar en **paquets** (**packages, biblioteques**). I els ***paquets es poden organitzar en *mòduls**, a partir de Java9.

Un **mòdul** és un conjunt de paquets interrelacionats amb un control de la visibilitat granular i més estricta que el mecanisme de visibilitat dels paquets.

Un **paquet** proporciona un nom i un mecanisme de control de la visibilitat, un paquet és un conjunt de classes, interfícies i recursos interrelacionats.

Amb l'ajuda dels paquets és possible donar el mateix nom a més d'una classe sempre i quan estiguin en paquets diferents.

20.3.1. Declaració de mòduls

TODO

20.3.2. Declaració de paquets

Característiques:

- Els **programes** de Java s'organitzen com a conjunts de paquets.
- Cada paquet té el seu conjunt de sub-paquets, que ajuden a l'organització de les classes i a evitar conflictes de nom.
- L'estructura nominal dels paquets és jeràrquica.
- El contingut d'un paquet poden ser classes, interfícies i sub-paquets.

- Els programes petits poden residir en un paquet sense nom o en un paquet amb un nom simple però si el paquet s'ha de distribuir els noms dels paquets **han de ser únics**.
- Si no s'especifica un paquet per una classe, la classe forma part del **paquet per defecte** anomenat **unnamed package**.

Per afegir una classe dins d'un paquet cal posar la següent declaració **com a primera instrucció del fitxer**.

```
package nom-del-paquet;
```

El nom del paquet pot consistir en una o més parts, per garantir la unicitat dels noms dels paquets **es sol utilitzar un nom de domini invertit**, per exemple, com.yahoo:

```
package common;
package local.test.utilitats;
```

Un fitxer de codi font pot tenir una única declaració *package*.



El nom d'una classe situada dins d'un paquet és: *nom-paquet.nom-classe*. Per exemple, el nom de la classe següent és **utils.Utilitat**:

```
package utils;

public class Utilitat {
    ...
}
```

Aquest nom s'anomena **FQN** o "Fully Qualified Name".

Com a conseqüència per executar la classe anterior, farfem **1.**

```
java utils.Utilitat
```

1 Suposant que la variable CLASSPATH estigui ben configurada

Col·lisió de noms

A part de per organitzar les biblioteques de classes els paquets proporcionen una manera de gestionar les **col·lisions de noms**.

Una col·lisió de noms és una situació en la que dues classes tenen el mateix nom.

Si diferents programadors creen dues classes amb el mateix nom però en paquets diferents, Java pot distingir-ne una de l'altra en base a la different ubicació en la estructura de paquets de l'aplicació.

Per exemple:

```
package paquet1;
public class ClasseA {
    ...
}

package paquet2;
public class ClasseA {
    ...
}
```

Ens podem referir a les classes anteriors com a:

```
paquet1.ClasseA a;
paquet2.ClasseA b;
```

20.3.3. El paquet per defecte

Existeix un paquet especial anomenat **default package** que no té nom.

Si no s'inclou cap clàusula *package* al principi d'un fitxer de codi font, la classe definida es posa automàticament al paquet per defecte.

20.4. Ubicació de les classes en paquets

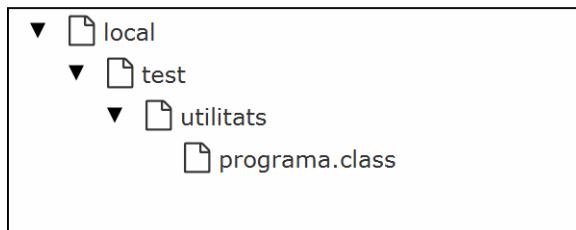
Un paquet es pot emmagatzemar en un sistema de fitxers específic, en una ubicació de xarxa o en una base de dades.



La ubicació física d'una classe dins d'un paquet ha de ser coherent amb el nom del paquet i la estructura de directoris que representa.

En un sistema de fitxers **cada part del nom del paquet correspon a un directori al host.**

Per exemple, el paquet *local.test.utilitats* indica l'existència d'una estructura de directoris com la següent:



El nom del paquet només especifica part de l'estructura de directoris on es troben els fitxers compilats, en l'exemple anterior, el directori local podria estar dins de la carpeta **C:\ProjectesJava**.

20.4.1. Com ho fa Java per trobar les classes definides per l'usuari

Conèixer el nom del paquet no és suficient per a localitzar un fitxer .class ja que indica una ruta relativa, la part faltant de la ruta s'ha d'obtenir des d'alguna altra font.

Per trobar les diferents classes definides per l'usuari Java pren com a referència una ruta anomenada **CLASSPATH** consistent en una llista de directoris, fitxers JAR, i fitxers ZIP que contenen fitxers de classe.

Un fitxer de classe té una ruta relativa que reflecteix el seu FQN. Per exemple, si la classe **com.mypackage.MyClass** està emmagatzemada dins de la carpeta /

myclasses aquesta carpeta hauria d'estar a la "class path" de l'usuari de manera que s'inclogui a la llista de directoris on es buscaran referències a les classes quan sigui necessari, si la classe està emmagatzemada dins d'un fitxer .jar anomenat **myclasses.jar**, aquest fitxer haurà d'estar a la "class path" de l'usuari.

El "class path" de l'usuari s'especifica com una cadena, separada per : en Linux i ; en Windows, amb totes les ubicacions possibles de les classes definides per l'usuari.

Aquestes ubicacions poden ser:

- Per defecte les classes es busquen al directori ".", és a dir al directori actual.
- El valor de la variable **CLASSPATH**, que sobreescriu el valor per defecte.
- El valor del paràmetre **-cp** o **-classpath** a la línia de comandes, que sobreescriu el valor per defecte i la variable CLASSPATH.
- El fitxer JAR especificat per l'opció **-jar** a la línia de comandes, que sobreescriu totes les altres opcions.

La variable d'entorn CLASSPATH

Mostrar el valor de la variable CLASSPATH:

Windows.

```
C:\> set CLASSPATH
```

Linux.

```
echo $CLASSPATH
```

Per eliminar el contingut de la variable CLASSPATH:

Windows.

```
C:\> set CLASSPATH=
```

Linux.

```
unset CLASSPATH; export CLASSPATH
```

Per establir el valor de la variable CLASSPATH:

Windows.

```
C:\> set CLASSPATH=C:\users\george\java\classes
```

Linux.

```
CLASSPATH=/home/george/java/classes; export CLASSPATH
```



També es pot definir un CLASSPATH temporal a l'hora d'executar un programa Java.

La comanda *java* admet el paràmetre **-cp** que permet establir un CLASSPATH per a l'execució del programa actual.

Per exemple, per establir el CLASSPATH a la carpeta actual:

```
java -cp . paquet.Test
```

Per establir el CLASPATH a la carpeta c:\tmp

```
java -cp c:\tmp paquet.Test
```

20.4.2. Declaracions import

Si volem utilitzar una classe ubicada dins d'un paquet podem utilitzar el seu nom FQN, per exemple:

```
java.util.Scanner in;  
in = new java.util.Scanner(System.in);
```

Però es pot simplificar el codi i fer-lo més lleigible utilitzant declaracions **import**.

En una declaració **import** es notifica al compilador de Java que es volen utilitzar una o més classes d'un paquet en particular. Això permet referir-se a les classes amb el seu nom simple i nom amb el FQN. Per exemple:

```
import java.util.Scanner;  
....  
Scanner in;  
in new Scanner(System.in);
```

No hi ha restriccions sobre el nombre de declaracions **import** que poden aparèixer en el codi font.

Sintaxis de les declaracions import:

- Una declaració import comença amb la paraula clau **import**.
- A continuació cal indicar el **nom d'un paquet** del qual es volen utilitzar classes en el codi font actual.

- Seguit del **nom d'una classe** o bé un * per indicar que es volen utilitzar una o més classes emmagatzemades dins del paquet indicat.
- Finalment una declaració import finalitza amb un **punt i coma**.

Per exemple, per utilitzar la classe **Persona** dins del paquet local.test.entitats caldrà la línia:

```
import local.test.entitats.Persona;
```

o bé

```
import local.test.entitats.*;
```



Les dues declaracions anteriors **no** inclouen classes dins dels paquets **local** o **local.test** .

20.5. Fitxers jar

Java proporciona l'eina **jar**² que permet comprimir tota una jerarquia de carpetes i classes en un únic fitxer d'extensió **.jar**. Això facilita enormement la distribució de les aplicacions.

Opcions de l'eina **jar**:

Opció	Descripció
c	Crea un nou fitxer.
C	Canvia els directoris durant l'execució de la comanda.
f	El primer element de la llista de fitxers és el nom del fitxer que es crearà.
i	Genera un fitxer amb un index de les classes i les seves ubicacions dins del fitxer jar.

² <http://docs.oracle.com/javase/8/docs/technotes/tools/windows/jar.html>

m	S'adjunta un fitxer de "manifest" extern.
M	No es crea el fitxer de manifest.
t	Mostra el contingut del fitxer jar.
u	Modifica el fitxer JAR.
v	"verbose", mostra informació addicional durant l'execució de l'eina.
x	Extreu els fitxes d'un fitxer JAR.
o	No utilitza compressió.

Exemples de d'ús de l'eina **jar**:

- Crea el fitxer JAR "Myjar.jar" amb tots els fitxers del paquet pack1 comprimits.

```
c:\javaprg> jar -cf Myjar.jar pack1
```

- **Extreure** el contingut d'un fitxer JAR:

```
c:\javaprg> jar -xf Myjar.jar
```

- **Mostrar** el contingut d'un fitxer JAR:

```
c:\javaprg> jar -tf Myjar.jar
```

- **Modificar** el contingut d'un fitxer JAR:

```
c:\javaprg>jar -uf Myjar.jar pack1
```

- Crear un JAR **executable**:

```
c:\javaprg>jar -cmf mainClass Myjar.jar pack1
```

mainClass és un fitxer de text, dins la carpeta **javaprg**, que conté:

```
"Main-Class: pack1.pack2.pack3.A"
```

on **pack1.pack2.pack3.A** és la classe on es troba el mètode **main** de l'aplicació.

- Executar un fitxer JAR:

```
c:\javaprg>java -jar Myjar.jar
```

20.6. Les biblioteques del llenguatge java

20.6.1. Biblioteques abans i després de Java8

Fins a la versió Java8 totes les biblioteques de Java es troben dins del JRE a partir de Java9 es separen les diferents biblioteques en mòduls traient del JRE per una banda tecnologies antigues, per exemple **java.corba**, i per l'altra biblioteques molt grans com per exemple **javafx**.

20.6.2. Mòduls de Java11

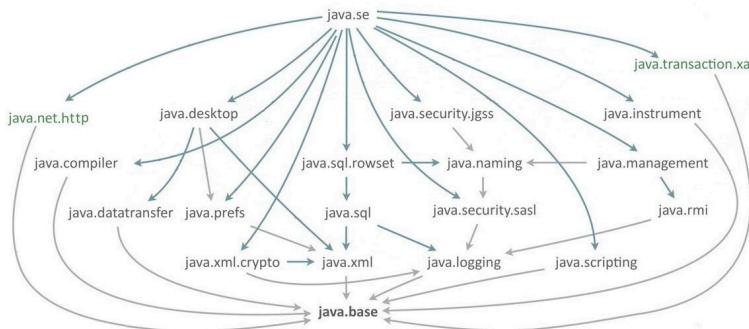
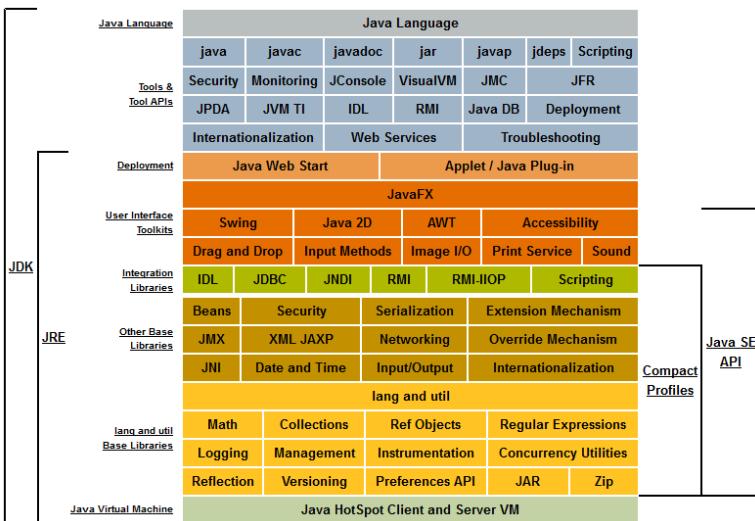


Figura 20.1. Mòduls de Java11

20.6.3. Biblioteques de Java8



El JDK ³ integra una sèrie de components ⁴ que configuren la plataforma de desenvolupament Java.

En particular la Java SE API ⁵ està configurada com una sèrie de llibreries ⁶ que encapsulen classes que proporcionen noves eines als programadors.

En aquest apartat comentarem breument alguns dels paquets de la api de Java.

20.6.4. java.lang

El paquet **java.lang** és especial. Conté les classes fonamentals que enllacen directament amb el llenguatge Java.

Conté algunes de les classes més utilitzades, per aquesta raó el paquet **java.lang** s'importa de forma automàtica a tots els fitxers de Java independentment d'haver afegit **import java.lang** o no.

Algunes de les classes que conté:

³ Java Development Kit

⁴ <https://docs.oracle.com/javase/8/docs/>

⁵ Application Programming Interface

⁶ <https://docs.oracle.com/javase/8/docs/api/>

java.lang.Math

Conté mètodes per realitzar operacions numèriques bàsiques, exponencials, trigonometria, logaritmes, arrels quadrades, etc...

totes les classes embolcall dels tipus primitius

Integer, Double, etc...

java.lang.String

Representa cadenes de caràcters.

java.lang.System

Conté mètodes d'utilitat general i d'interacció amb el sistema host.

20.6.5. java.util

Aquesta biblioteca proporciona utilitats genèriques de Java com ara **coleccions**, la classe **Scanner**, utilitats de **manipulació de matrius**, el model d'**esdeveniments**, utilitats de **data i hora i internacionalització**.

20.6.6. java.util.Arrays

En particular la classe `java.util.Arrays` conté mètodes útils per ajudar amb les operacions més comuns amb les matrius.

Tots aquests mètodes funcionen per a tots els tipus primitius.

Mètode sort

El mètode `sort` permet ordenar una matriu o part d'una matriu.

Per exemple, el codi següent ordena una matriu de nombres i una matriu de caràcters.

```
double[] nombres = {6.0, 4.4, 1.9, 2.9, 3.4, 3.5};  
java.util.Arrays.sort(nombres); // Ordenem la matriu sencera  
  
char[] chars = {'a', 'A', '4', 'F', 'D', 'P'};  
java.util.Arrays.sort(chars, 1, 3); // Ordenem els elements segon i  
tercer de la matriu
```

Mètode binarySerach

El mètode `binarySearch` permet buscar un valor dins d'una matriu.

La matriu ha d'estar ordenada ascendentment perquè el mètode funcioni correctament.

Per exemple:

```
int[] list = {2, 4, 7, 10, 11, 45, 50, 59, 60, 66, 69, 70, 79};
System.out.println("1. index " +
java.util.Arrays.binarySearch(list, 11));
System.out.println("2. index " +
java.util.Arrays.binarySearch(list, 12));
char[] chars = {'a', 'c', 'g', 'x', 'y', 'z'};
System.out.println("3. index " +
java.util.Arrays.binarySearch(chars, 'a'));
System.out.println("4. index " +
java.util.Arrays.binarySearch(chars, 't'));
```

```
1. index 4
2. index -6
3. index 0
4. index -4
```

Mètode equals

El mètode *equals* permet determinar si dues matrius tenen el mateix contingut.

Per exemple:

```
int[] list1 = {2, 4, 7, 10};
int[] list2 = {2, 4, 7, 10};
int[] list3 = {4, 2, 7, 10};
System.out.println(java.util.Arrays.equals(list1, list2)); // true
System.out.println(java.util.Arrays.equals(list2, list3)); // false
```

Mètode fill

El mètode *fill* permet omplir tota o part d'una matriu.

Per exemple:

```
int[] list1 = {2, 4, 7, 10};
int[] list2 = {2, 4, 7, 10};
```

```
java.util.Arrays.fill(list1, 5); // Omple tota la matriu de 5
java.util.Arrays.fill(list2, 1, 3, 8); // Omple l'element 1 i 2 de 8
```

Mètode **toString**

El mètode **toString** retorna una cadena amb una representació de tots els elements de la matriu. És una manera fàcil i simple de mostrar el contingut d'una matriu.

Per exemple:

```
int[] list = {2, 4, 7, 10};
System.out.println(Arrays.toString(list));
```

[2, 4, 7, 10].

20.6.7. **java.util.ArrayList**

Les matrius de Java no permeten modificar la seva longitud, no obstant les biblioteques de Java proporcionen la classe **ArrayList**.

Un objecte de tipus **ArrayList** es pot utilitzar com una matriu de longitud variable.

Una **diferència important** entre una **ArrayList** i una matriu és que la primera **només pot emmagatzemar tipus per referència**, no pot emmagatzemar tipus primitius.

Si es vol treballar amb tipus primitius caldrà declarar l'**ArrayList** utilitzant alguna de les **classes embolcall** apropiada. Per exemple, per declarar una **ArrayList** d'elements enters caldria declarar l'**ArrayList** com una col·lecció d'elements **Integer**.

Per declarar un objecte de tipus **ArrayList** s'utilitza la següent sintaxi:

```
ArrayList<tipus_de_dades> nom_de_variable;
```

on **tipus_de_dades** indica quin és el tipus de dades dels elements que conté la col·lecció.

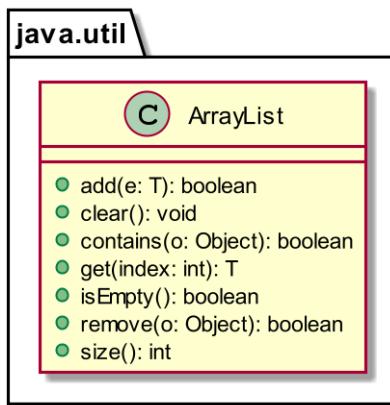
Per crear un objecte de tipus **ArrayList** s'utilitza la sintaxi següent:

```
nom_de_variable = new ArrayList<>();
```

El següent exemple mostra la creació d'un objecte de tipus **ArrayList**:

```
ArrayList<String> = new ArrayList<String>();
```

A continuació es mostren alguns dels mètodes més comuns de la classe **ArrayList**:



La **T** que apareix com a tipus de dades en alguns dels mètodes de la classe **ArrayList** indica el tipus de dades que conté la col·lecció.

I un exemple d'utilització dels mètodes anteriors:

```

import java.util.ArrayList;

public class ArrayListTest {

    public static void main(String[] args) {

        ArrayList<Integer> llista = new ArrayList<>();

        int nombreElements = llista.size(); // En aquest punt total val
        0
    }
}

```

```

        System.out.println("La mida de la ArrayList és " +
nombreElements);
        System.out.println("Els elements de la ArrayList són " +
llista.toString());

        llista.add(new Integer(10));
        llista.add(20); // Autoboxing
        llista.add(30); // Autoboxing

        nombreElements = llista.size();

        System.out.println("La mida de la ArrayList és " +
nombreElements);
        System.out.println("Els elements de la ArrayList són " +
llista.toString());

        llista.clear();

        nombreElements = llista.size(); // En aquest punt total val 0

        System.out.println("La mida de la ArrayList és " +
nombreElements);
        System.out.println("Els elements de la ArrayList són " +
llista.toString());
    }
}

```

20.6.8. *java.time*

Els tipus definits dins d'aquesta biblioteca representen conceptes relacionats amb dates i temps com per exemple, instants, dureacions, dates, hores, períodes, zones-temporals, etc...

Exemples d'algunes de les classes que es troben a aquesta biblioteca són:

LocalDate

Estructura que emmagatzema una data sense la hora.

LocalTime

Estructura que emmagatzema una hora sense la data.

LocalDateTime

Estructura que emmagatzema una data i una hora.

20.6.9. *java.io*

Aquesta biblioteca conté les classes que gestionen les interaccions bàsiques d'entrada i sortida en Java. Les classes d'entrada/sordida d'aquest paquet es poden agrupar de la següent manera:

- Classes per llegir des d'un flux de dades.
- Classes per escriure a un flux de dades.
- Classes per manipular fitxers en un sistema de fitxers local.
- Classes per gestionar la serialització d'un objecte.

20.6.10. *java.net*

El paquet *java.net* conté classes i interfícies que proporcionen una potent infraestructura de xarxa en Java.

Essencialment hi ha classes de enfocades a dos nivells:

- Classes que treballen amb la xarxa a capa 1, 2, 3 i 4.
- Classes que treballen a capa 5, 6 i 7.

20.6.11. *java.security*

Aquest paquet aporta diverses classes que permeten encriptar fitxers, treballar amb signatures digitals i amb criptografia.

20.6.12. *java.sql*

Conté totes les classes i interfícies que proporcionen una API per accedir i processar dades emmagatzemades en un origen de dades.

20.6.13. *java.swing*

És una llibreria per desenvolupar aplicacions Java d'escriptori.

20.6.14. *javafx*

Les biblioteques de JavaFX proporcionen una API pel desenvolupament d'**aplicacions gràfiques d'escriptori** i aplicacions de tipus **rich Internet Applications** o aplicacions RIA.

21

Alguns problemes difícils

Alguns programes són difícils de resoldre amb les tècniques que coneixem fins al moment.

Per introduir la següent secció intentarem resoldre amb les eines que tenim els següents dos problemes.

21.1. Implementar l'eina "rellenar"

La majoria de programes d'edició d'imatges disposen d'una eina "fill" que permet "omplir" d'un color una àrea normalment delimitada per línies.

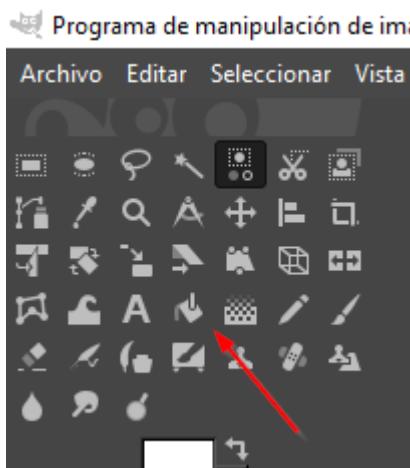


Figura 21.1. Eina "rellenar" al programa d'edició d'imatges GIMP

Estem interessats en l'algoritme que hi ha al darrere d'aquesta eina i per analitzar-lo considerem el següent problema més elemental:

El següent codi genera una matriu de dues dimensions i l'omple amb dos caràcters, una **X** i una **O**. Volem que l'usuari pugui triar una de les coordenades de la matriu i el programa canviï el símbol seleccionat i tots els símbols adjacents, ortogonalment, a aquests pel caràcter ..

Per exemple:

```

0 1 2 3 4 5 6 7 8 9
0 0 0 X 0 0 0 X X X X
1 0 0 0 0 0 0 X 0 X X
2 0 X X X X 0 X 0 0 0
3 0 0 X 0 0 0 X X X X
4 X 0 0 0 X X 0 0 0 0
5 X 0 X X X 0 0 0 0 X
6 X 0 X X X X X X X X
7 X 0 X X X X X X X X
8 X 0 0 0 X 0 0 0 0 X
9 X X X X 0 0 0 0 0 0

```

Introduceix una fila: 5

Introduceix una columna: 2

```

0 1 2 3 4 5 6 7 8 9
0 0 0 X 0 0 0 X X X X
1 0 0 0 0 0 0 X 0 X X
2 0 X X X X 0 X 0 0 0
3 0 0 X 0 0 0 X X X X
4 X 0 0 0 . . 0 0 0 0
5 X 0 . . . 0 0 0 0 .
6 X 0 . . . . . . .
7 X 0 . . . . . . .
8 X 0 0 0 . 0 0 0 0 .
9 X X X X 0 0 0 0 0 0

```

```

package buscazoness;

import java.util.Scanner;

public class BuscaZones {

    private static final int ZONA1 = 1;
    private static final int ZONA2 = 0;
    private static final char ZONA1_SPRITE = 'X';
}

```

```
private static final char ZONA2_SPRITE = '0';
private static final char ZONA3_SPRITE = '.';
private static char[][] mapa;

public static void main(String[] args) {
    mapa = generarMapa(new String[]{
        "0010001111",
        "0000001011",
        "0111101000",
        "0010001111",
        "1000110000",
        "1011100001",
        "1011111111",
        "1011111111",
        "1000100001",
        "1111000000"});
}

mostrarMapa();

Scanner in = new Scanner(System.in);

System.out.println();
System.out.print("Introdueix una fila: ");
int fila = in.nextInt();
System.out.print("Introdueix una columna: ");
int columna = in.nextInt();
System.out.println();

// TODO: Dibuixar tota la zona adjacent a (fila, columna) amb
ZONA3_SPRITE
mostrarMapa();
}

public static void mostrarMapa() {
    System.out.print("  ");
    for (int j = 0; j < mapa[0].length; j++) {
        System.out.printf("%2d", j);
    }
    System.out.println();

    for (int i = 0; i < mapa.length; i++) {
        System.out.printf("%2d ", i);
        for (int j = 0; j < mapa[0].length; j++) {
            System.out.print(" " + mapa[i][j]);
        }
    }
}
```

```

        System.out.println();
    }

}

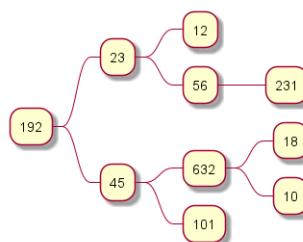
public static char[][] generarMapa(String[] sLaberint) {
    char[][] retorn = new char[sLaberint.length]
[sLaberint[0].length()];
    for (int i = 0; i < sLaberint.length; i++) {
        for (int j = 0; j < sLaberint[i].length(); j++) {
            switch
(Character.getNumericValue(sLaberint[i].charAt(j))) {
                case ZONA1:
                    retorn[i][j] = ZONA1_SPRITE;
                    break;
                case ZONA2:
                    retorn[i][j] = ZONA2_SPRITE;
                    break;
            }
        }
    }
    return retorn;
}
}

```

21.2. Realitzar cerques en una estructura de directoris

Imaginem que volem implementar una eina similar a la comanda **find** de Linux que ens permeti trobar un fitxer dins de l'estructura de fitxers en base a una expressió regular per exemple.

Per analitzar l'algoritme subjacent atacarem un problema més senzill, el següent codi crea una estructura en arbre com la mostrada en la següent imatge:



Considerem que treballem només amb arbres els nodes dels quals poden tenir cap, un o dos nodes com a màxim i pensem que cada node pot emmagatzemar un número enter, el codi que es proporciona a continuació fa precisament això.

Volem implementar una funció que sigui capaç de mostrar tots els elements de l'arbre, independentment de la seva mida, i una funció que retorni l'element major de l'arbre.

```
package cercaenarbre;

class Node {

    int valor;
    Node fill1;
    Node fill2;
}

public class CercaEnArbre {

    // 192
    // |-- 23
    //     +- 12
    //     +- 56
    //         +- 231
    // +- 45
    //     +- 632
    //         +- 18
    //         +- 10
    //         +- 101
    public static void main(String[] args) {
        Node arrel = crearNode(192);
        arrel.fill1 = crearNode(23);
        arrel.fill1.fill1 = crearNode(12);
        arrel.fill1.fill2 = crearNode(56);
        arrel.fill1.fill1.fill1 = crearNode(231);
        arrel.fill2 = crearNode(45);
        arrel.fill2.fill1 = crearNode(632);
        arrel.fill2.fill1.fill1 = crearNode(18);
        arrel.fill2.fill1.fill2 = crearNode(10);
        arrel.fill2.fill2 = crearNode(101);

        // TODO: Escriure una funció que sigui capaç de mostrar tots els
        // nodes de l'arbre
    }
}
```

```
// TODO: Escriure una funció que retorni el major element de
l'arbre.
}

private static Node crearNode(int valor) {
    Node n = new Node();
    n.valor = valor;
    n.fill1 = null;
    n.fill2 = null;

    return n;
}

}
```

22

Recursivitat

La recursivitat és una tècnica que proporciona solucions elegants a problemes que són difícils de resoldre mitjançant bucles.

Utilitzar la recursivitat és programar utilitzant mètodes recursius, això és, **mètodes que es criden a si mateixos**.

La recursivitat és una tècnica útil, en alguns casos permet trobar solucions simples a problemes complicats.

22.1. Implementació interna en un ordinador

Recordeu que quan es crida a una funció, el sistema crea un registre d'activació que emmagatzema els paràmetres i les variables locals pel mètode. Aquest registre s'emmagatzema a una àrea de memòria anomenada **pila de crides a funcions**.

Quan un mètode crida a un altre mètode, el registre d'activació del mètode inicial es manté intacte i un nou registre d'activació es crea a la pila pel nou mètode cridat.

Quan un mètode acaba i retorna a qui l'ha cridat, el registre d'activació del mètode finalitzat s'elimina de la pila, i per tant totes les variables locals de tipus primitius totes les referències a objectes s'eliminen.

Si un mètode es crida a si mateix es van creant nous registres d'activació a la pila **que mantenen les variables locals de cadascuna de les crides al propi mètode**.

22.2. Algoritmes recursius

Per què un algoritme recursiu funcioni correctament cal que compleixi tres regles:

1. Existeix un **cas base** pel qual l'algoritme finalitza sense recursió.
2. Tots els arguments de les crides recursives es reenvien tendint al cas base.
3. Cada crida a la funció recursiva es realitza sobre un problema de menor complexitat.



Si no existeix un **cas base** o s'hi arriba mai, la pila creixerà indefinidament i es produirà un error de tipus **StackOverflowError**.

```
public static void infinit(String s) {
    System.out.println(s);
    infinit(s);
}
```

22.3. Exemple - comptaEnrrera

Considerem el següent programa:

Exemple de programa recursiu.

```
public static void comptaEnrrera(int n) {
    if (n == 0) { ❶
        System.out.println("Zero!!");
    } else {
        System.out.println(n);
        comptaEnrrera(n - 1); ❷ ❸
    }
}
```

- ❶ Cas base.
- ❷ Al passar **n-1** estem tendint al cas base **n=0** assegurant la finalització de l'algoritme.
- ❸ És més fàcil comptar endarrere des de **n-1** que des de **n**, és a dir, la crida recursiva s'ha de definir sobre un problema d'emenor complexitat.

22.4. Exemple - mostrarLínies

Aquest mètode agafa un enter, n, com a paràmetre i mostra n línies. Mentre n sigui positiu mostra una línia en blanc i es crida a si mateixa per a mostrar n-1 línies més.

Exemple de programa recursiu.

```
public static void nLinies(int n) {
    if (n > 0) { ①
        System.out.println();
        nLinies(n - 1); ② ③
    }
}
```

- ① Cas base (**n=0**).
- ② Al passar **n-1** estem tendint al cas base **n=0** assegurant la finalització de l'algoritme.
- ③ És més fàcil pintat **n-1** línies que **n**.

22.5. Exemple - Càlcul del factorial d'un número

Volem un programa que calculi el factorial d'un número $n \geq 0$.

Per enfocar-ne la implementació recursiva ens fixem en dues característiques de la funció factorial.

1. $0! = 1$
2. $n! = n \cdot (n - 1)!$

El punt 2 ens dona la recursivitat i el punt 1 ens dona el cas base.

Per tant, es pot definir la funció *factorial* com:

```
public static int factorial(int n) {
    if (n == 0) { ①
        return 1;
    }
    int recursiu = factorial(n - 1); ②
```

```

int resultat = n * recursiu; ③
return resultat;
}

```

- ① Cas base (**n=0**).
- ② Al passar **n-1** estem tendint al cas base **n=0** assegurant la finalització de l'algoritme.
- ③ $(n-1)!$ és menys complex que $n!$.



És més simple i més eficient calcular el factorial de n amb un bucle.

22.6. Exemple - Successió de Fibonacci

La successió de Fibonacci és una successió matemàtica de nombres naturals tal que **cada un dels seus termes és igual a la suma dels dos anteriors**.

Aquesta successió fou descrita per primera vegada per **Leonardo de Pisa Fibonacci** i cadascun dels seus termes rep el nom de nombre de Fibonacci.

Els primers 20 termes de la successió de Fibonacci.

1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	1597
2584	4181	6765														

En primer lloc establím una definició recursiva de la successió:

```

fibonacci(1) = 1
fibonacci(2) = 1
fibonacci(n) = fibonacci(n - 1) + fibonacci(n-2)

```

i per tant:

```

public static int fibonacci(int n) {
    if (n == 1 || n == 2) {
        return 1;
    }
    return fibonacci(n - 1) + fibonacci(n - 2);
}

```



És més eficient calcular la successió de fibonacci amb bucles.

22.7. Exemple - Palindroms

Un palíndrom és una paraula que es llegeix igual del dret que del revés.

El problema de detectar si una paraula és un palíndrom es pot dividir en dos subproblemes:

1. Comprovar si la primera i la última lletra de la paraula són iguals.
2. Ignorar la primera i la última lletra i mirar si la resta de la paraula és un palíndrom.

El segon problema és com l'original però més simple, això fa el problema de detectar palindroms candidat a resoldre'l recursivament.

En aquest problema hi ha dos **casos base**:

1. El primer i l'últim caràcter no coincideixen i la cadena no és un palíndrom.
2. La mida de la cadena és 0 o 1 i la cadena si que és un palíndrom.

Versió poc eficient utilitzant el mètode substring.

```
static boolean esPalindrom(String s) {
    if (s.length() <= 1) {
        return true;
    } else if (s.charAt(0) != s.charAt(s.length() - 1)) {
        return false;
    } else {
        return esPalindrom(s.substring(1, s.length() - 1));
    }
}
```

La versió anterior del mètode esPalindrom **no és eficient** perquè crea una nova cadena per a cada crida recursiva. Per evitar crear noves cadenes es pot reimplementar el mètode accedint manualment a les parelles de caràcters que es volen comparar amb el mètode *charAt*.

```
static boolean esPalindrom(String s) {
```

```

    return esPalindrom(s, 0, s.length() - 1);
}

static boolean esPalindrom(String s, int low, int high) {
    if (high <= low) { // Cas base
        return true;
    } else if (s.charAt(low) != s.charAt(high)) { // Cas base
        return false;
    } else {
        return isPalindrome(s, low + 1, high - 1);
    }
}

```

22.8. Exemple - cercaBinaria

La cerca binària funciona en **matrius ordenades**.

1. La cerca binària comença per comparar l'element central de la matriu amb el valor buscat.
2. Si el valor buscat és igual a l'element central, es retorna la seva posició.
3. Si el valor buscat és menor o major que l'element central, la cerca contínua en la primera o segona meitat, respectivament, deixant l'altra meitat fora de consideració.

```

package recursivitat;

public class Recursivitat {

    public static int cercaBinaria(int[] matriu, int indexMin, int
indexMax, int valorBuscat) {
        if (indexMax >= 0 && matriu[indexMin] <= valorBuscat &&
matriu[indexMax] >= valorBuscat) {
            int mig = obtenirValorMig(indexMin, indexMax);
            System.out.println(String.format("(%d, %d) valor mig:
m[%d]=%d, valor a buscar: %d", indexMin, indexMax, mig, matriu[mig],
valorBuscat));
            if (matriu[mig] == valorBuscat) {
                return mig;
            } else if (matriu[mig] < valorBuscat) {
                return cercaBinaria(matriu, mig + 1, indexMax,
valorBuscat);
            }
        }
    }
}

```

```

        return cercaBinaria(matriu, indexMin, mig - 1, valorBuscat);
    }
    return -1;
}

public static int obtenirValorMig(int minLimit, int maxLimit) {
    return (maxLimit + minLimit) / 2;
}

public static void main(String[] args) {
    int value = 5;
    int[] matriu =
{1, 2, 5, 6, 10, 11, 67, 89, 104, 170, 345, 1002, 2334, 5000};
    int indexResultat = cercaBinaria(matriu, 0, matriu.length - 1,
value);
    System.out.println(String.format("Resultat %d", indexResultat));
}
}

```

22.9. Conveniència de l'ús de la recursividat

La recursió s'hauria d'evitar sempre que es pugui per motius d'eficiència. No obstant això, pot ser justificable fer-la servir quan:

1. Se sap que la funció no generarà massa profunditat en les crides recursives.
2. Se sap que la funció no utilitzarà estructures de dades locals massa grans.
3. Les crides no generen crides ja resoltes. Aquest és un problema d'ineficiència freqüent en la recursivitat.
4. La solució no és plantable de forma senzilla de cap altre manera. Si la solució iterativa és més clara es recomana utilitzar aquesta versió.

22.10. Backtracking

El Backtracking és una tècnica de resolució general de problemes mitjançant una cerca sistemàtica de solucions.

El procediment general es basa en la descomposició del procés de cerca en tasques parcials de tanteig de solucions (trial and error).

La solució d'una tasca parcial permet construir la solució del problema de partida. Les tasques parcials són similars a la de partida i es poden resoldre directament

o tornar a descompondre en altres tasques parcials. Les tasques parcials es plantegen de forma recursiva en construir gradualment les solucions.

Per a resoldre cada tasca, es descompon en diverses subtasques i es comprova si alguna d'elles condueix a la solució del problema. Les subtasques es proven una a una, i si alguna d'elles condueix a la solució, el problema està resolt, en cas contrari el problema no té solució.

La descripció natural del procés es representa per un arbre de cerca en el qual es mostra com cada tasca es ramifica en subtasques. L'arbre de cerca acaba en fulles on es troben solucions o on es conclou que per aquesta branca no és possible trobar-la.

23

Introducció a Processing

Processing (<https://processing.org>) és un llenguatge de programació basat en Java i de codi obert. El llenguatge està dissenyat per la producció de projectes multimèdia i interactius de disseny digital.

23.1. Mètode setup()

La funció **setup()** s'executa una sola vegada quan arranca el programa. S'utilitza per inicialitzar les propietats inicials de l'entorn com la mida de la pantalla, el **framerate** i la càrrega d'imatges i fonts.

23.2. Mètode draw()

El mètode **draw()** s'executa en bucle fins que el programa s'atura. El nombre de vegades que s'executa la funció es pot controlar amb el mètode **frameRate()**.

23.3. Paràmetres típics d'inicialització

Funció size()

Estableix la mida de la pantalla.

Funció fullScreen()

Executa l'aplicació a pantalla completa.

Funció frameRate()

Estableix la quantitat de frames per segon de l'aplicació.

Variable width

Manté l'amplada de la pantalla de l'aplicació.

Variable height

Manté l'alcada de la pantalla de l'aplicació.

Per exemple:

```
void setup()
{
    size(800, 600);
    frameRate(60);
}
```

23.4. Coordenades

La coordenada (0, 0) es troba a l'extrem superior esquerra de la pantalla. L'eix de les **x** creix en positiu cap a la dreta. L'eix de les **y** creix en positiu cap avall.

23.5. Formes bàsiques

Funció rect()

Dibuixa un rectangle.

Funció ellipse()

Dibuixa una el·lipse.

fill()

Determina el color de les següent formes.

noFill()

Dibuixa les formes sense pintar.

stroke()

Determina el color de la vora de la forma.

noStroke()

Dibuixa les formes sense vora.

background()

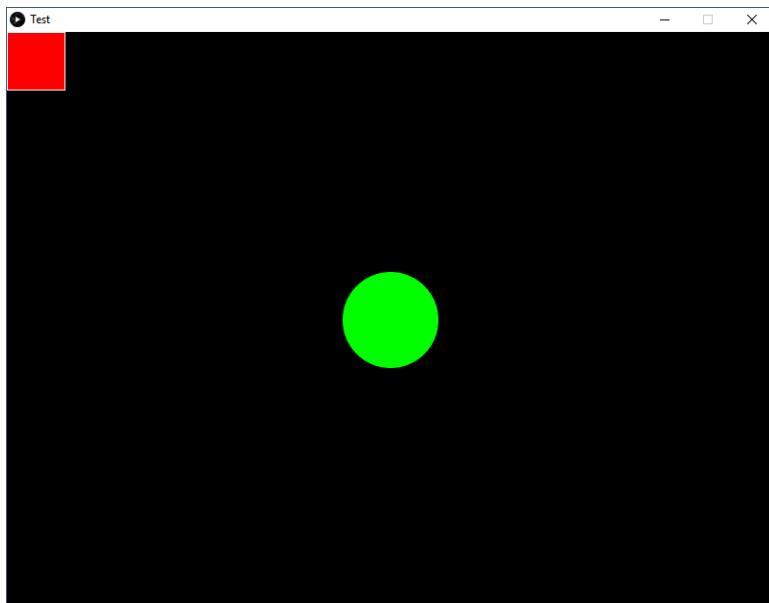
Pinta la pantalla del color especificat.



Es poden pintar més formes bàsiques, mirar la referència del llenguatge.

Per exemple:

```
void setup() {  
    size(800, 600);  
    frameRate(60);  
}  
  
void draw() {  
    background(0, 0, 0);  
    fill(255, 0, 0);  
    stroke(255, 255, 255);  
    rect(0, 0, 60, 60);  
    noStroke();  
    fill(0, 255, 0);  
    ellipse(400, 300, 100, 100);  
}
```



23.6. Mètode *keyPressed()*

El mètode **keyPressed()** permet detectar la pulsació de tecles per part de l'usuari.

Variable key

Manté el *char* de la última tecla polsada.

Exemple:

```
private int posX;
private int posY;

void setup() {
    size(800, 600);
    frameRate(60);
    posX = 400;
    posY = 300;
}

void draw() {
    background(0, 0, 0);
    fill(0, 255, 0);
    ellipse(posX, posY, 100, 100);
}

void keyPressed() {
    switch(key) {
        case 'w':
            posY-=3;
            break;
        case 'd':
            posX+=3;
            break;
        case 's':
            posY+=3;
            break;
        case 'a':
            posX-=3;
            break;
    }
}
```

23.7. Mètode *mousePressed()*

El mètode **mousePressed()** permet detectar la pulsació de tecles per part de l'usuari.

Variable mouseButton

Manté l'últim botó premut del ratolí.

Variable mouseX

Manté la posició sobre l'eix x del ratolí.

Variable mouseY

Manté la posició sobre l'eix y del ratolí.

Exemple:

```
private int posX;
private int posY;

void setup() {
    size(800, 600);
    frameRate(60);
    posX = 400;
    posY = 300;
}

void draw() {
    background(0, 0, 0);
    fill(0, 255, 0);
    ellipse(posX, posY, 100, 100);
}

void mousePressed() {
    posX = mouseX;
    posY = mouseY;
}
```

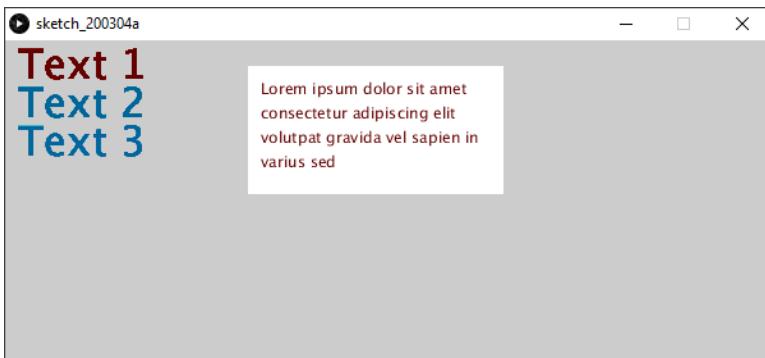
23.8. Mostrar text

Funció text()

Mostra una cadena per pantalla a una posició donada. Es pot especificar una zona rectangular perquè el text hi quedí enmarcat.

Funció textSize()

Determina la mida del text a mostrar.



```
void setup() {
    size(600, 250);
}

void draw() {
    textSize(32);
    text("Text 1", 10, 30);
    fill(0, 102, 153);
    text("Text 2", 10, 60);
    fill(0, 102, 153, 51);
    text("Text 3", 10, 90);

    String s = "Lorem ipsum dolor sit amet consectetur adipiscing elit
volutpat gravida vel sapien in varius sed";
    textSize(12);
    fill(255, 255, 255);
    noStroke();
    rect(190, 20, 200, 100);
    fill(100, 0, 0);

    text(s, 200, 30, 200, 100);
}
```

23.9. Mostrar imatges

Classe PImage

És el tipus de dada que permet encapsular imatges.

Funció loadImage()

Retorna un objecte PImage a partir del seu nom de fitxer.

Funció image()

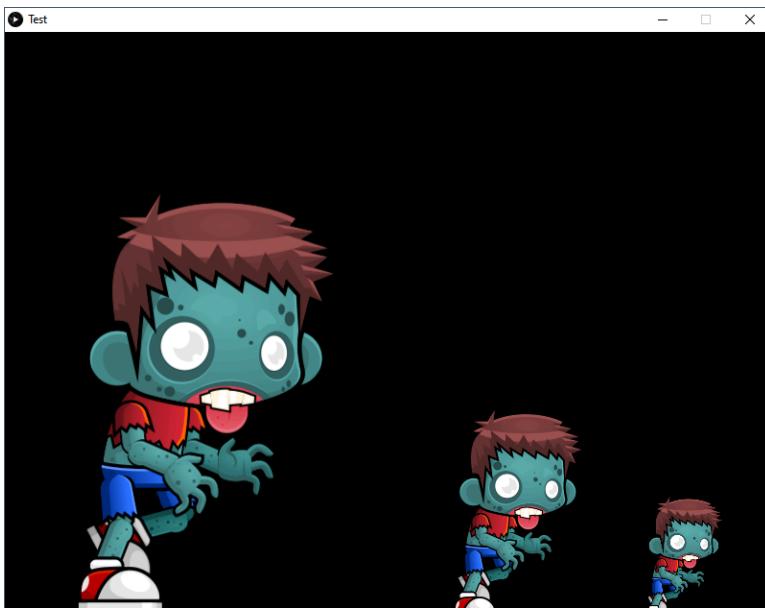
Permet mostrar una imatge carregada dins d'un objecte PImage. També permet mostrar una imatge reescalada.

```
private int posX;
private int posY;
private PImage zombie;

void setup() {
    size(800, 600);
    frameRate(60);
    zombie = loadImage("images/zombie.png");
    posX = 400;
    posY = 300;
}

void draw() {
    background(0, 0, 0);
    int w=zombie.width;
    int h=zombie.height;

    image(zombie, 0, 100, w, h);
    image(zombie, w, 100+h*0.5, w*0.5, h*0.5);
    image(zombie, w+w*0.5, 100+h*0.7, w*0.3, h*0.3);
}
```



23.10. Temps

La funció **millis()** retorna la quantitat de milisegons que han passat des de l'arrencada del programa.

És útil per controlar animacions que depenen del temps.

Per exemple el següen codi dibuixa una circumferència cada segon.

```
private long millisSeguentCercle;

void setup() {
    size(600, 480);
    frameRate(60);
    background(0, 0, 0);
}

void draw() {
    if (millisSeguentCercle<=millis()) {
        fill(255, 255, 255);
        ellipse(random(width), random(height), 50, 50);
        millisSeguentCercle=millis()+1000;
    }
}
```

23.11. Animacions

Per fer animacions n'hi ha prou en mostrar els diferents fotogrames a cada frame. El problema és que la velocitat de l'animació depèn del **framerate** establert.

Podeu veure-ho al següent exemple:

```
PIImage[] images = new PImage[10];
int currentFrame;
int numFrames = 10;

void setup() {
    size(400, 600);

    loadImages();
}

void draw() {
    background(0);

    currentFrame = (currentFrame+1) % numFrames;
    PImage currentImage = images[(currentFrame) % numFrames];
    image(currentImage, 0, height - currentImage.height);
}

void loadImages() {
    images[0] = loadImage("Walk_000.png");
    images[1] = loadImage("Walk_001.png");
    images[2] = loadImage("Walk_002.png");
    images[3] = loadImage("Walk_003.png");
    images[4] = loadImage("Walk_004.png");
    images[5] = loadImage("Walk_005.png");
    images[6] = loadImage("Walk_006.png");
    images[7] = loadImage("Walk_007.png");
    images[8] = loadImage("Walk_008.png");
    images[9] = loadImage("Walk_009.png");
}
```

Si volem no dependre del **framerate** podem basar l'animació en el temps com es mostra en el següent exemple:

```
PIImage[] images = new PImage[10];
int currentFrame;
```

```

int numFrames = 10;
int timeToNextFrame = 0;

void setup() {
    size(400, 600);
    loadImages();
}

void draw() {
    background(0);

    if (timeToNextFrame < millis()) {
        currentFrame = (currentFrame+1) % numFrames;
        timeToNextFrame = millis() + 200;
    }
    PImage currentImage = images[(currentFrame) % numFrames];
    image(currentImage, 0, height - currentImage.height);
}

void loadImages() {
    images[0] = loadImage("Walk_000.png");
    images[1] = loadImage("Walk_001.png");
    images[2] = loadImage("Walk_002.png");
    images[3] = loadImage("Walk_003.png");
    images[4] = loadImage("Walk_004.png");
    images[5] = loadImage("Walk_005.png");
    images[6] = loadImage("Walk_006.png");
    images[7] = loadImage("Walk_007.png");
    images[8] = loadImage("Walk_008.png");
    images[9] = loadImage("Walk_009.png");
}

```

23.12. Control de col·lisions

```

class Bola {
    int x;
    int y;
    int vel;
    int diametre = 50;
    color c;
}

Bola b1;
Bola b2;

```

```
boolean colision = false;

void setup() {
    size(800, 200);
    b1 = new Bola();
    b1.x = 0;
    b1.y = 100;
    b1.c = color(255, 0, 0);
    b1.vel = 1;

    b2 = new Bola();
    b2.x = width;
    b2.y = 100;
    b2.c = color(255, 255, 0);
    b2.vel = -2;
}

void draw() {
    detectarColisio(b1, b2);

    if (colision==false) {
        background(255, 255, 255);
    } else {
        background(0, 0, 0);
    }

    b1.x += b1.vel;
    dibuixarBola(b1);

    b2.x += b2.vel;
    dibuixarBola(b2);
}

void dibuixarBola(Bola b) {
    fill(b.c);
    ellipse(b.x, b.y, b.diametre, b.diametre);
}

void detectarColisio(Bola a, Bola b) {
    if (a.diametre/2+b.diametre/2<= dist(a.x, a.y, b.x, b.y)) {
        colision = true;
    } else {
        colision = false;
    }
}
```

}

Bibliografia

Llibres

Allen B. Downey. 'Think Java'. O'Reilly Media. 2016. ISBN 978-1-491-92956-8

Kishori Sharan. 'Beginning Java 8 Fundamentals'. Apress. 2014. ISBN 978-1-430-26653-2.

Robert C. Martin. 'Código limpio: Manual de estilo para el desarrollo ágil de software'. Anaya Multimedia. 2012. ISBN 978-8-441-53210-6

Daniel Liang. 'Introduction to Java Programming, Comprehensive Version'. 9th Edition. Pearson. 2012. ISBN 978-0-132-93652-1

Jan Vantomme. 'Processing 2: Creative Programming Cookbook'. Packt Publishing. 2012. ISBN 978-1-849-51794-2

Robert Liguori. 'Java 8 Pocket Guide'. O'Reilly Media. 2014. ISBN 978-1-491-90086-4

Allan Vermeulen. 'The Elements of Java Style'. 9th Edition. Cambridge University Press. 2007. ISBN 978-0-521-77768-1.

Stuart Reges. 'Building Java Programs: A Back to Basics Approach' 2nd Edition. Pearson. 2010. ISBN 978-0-136-09181-3

Walter J. Savitch. 'Java: An Introduction to Problem Solving & Programming'. 6th Edition. Pearson. 2011. ISBN 978-0-132-16270-8

Robert Sedgewick. 'Algorithms'. 4th Edition. Addison-Wesley Professional. 2011. ISBN 978-0-321-57351-3

Joshua Bloch. 'Effective Java'. 2nd Edition. Addison-Wesley. 2008. ISBN 978-0-321-35668-0

Part III. UF3 Fonaments de gestió de fitxers

Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots. So far, the Universe is winning.

— Rich Cook *escriptor de novel·les de ciència ficció*

Sumari

24. Resultats d'aprenentatge i criteris d'avaluació	239
25. Continguts	241
26. Gestió de fitxers	243
26.1. Entrada / Sortida	243
26.2. Paquets java.io i java.nio	243
26.3. Conèixer el directori de treball	244
26.4. "Paths" o rutes	245
26.5. Tractament d'errors en l'accés a fitxers	246
26.6. La classe File	247
26.7. Creació d'un objecte File	251
26.8. Comprovar si el fitxer existeix	252
26.9. Comprovar si el fitxer és un directori	252
26.10. Ruta absoluta i ruta canònica	252
26.11. Crear, renombrar i eliminar fitxers	254
26.12. Treballar amb els atributs dels fitxers	256
26.13. Copiar un fitxer	257
26.14. Saber la mida d'un fitxer	257
26.15. Mostrar tots els directoris arrel	257
26.16. Mostrar tots els fitxers i directoris d'un directori	258
27. Lectura i escriptura de fitxers	261
27.1. Classes de la biblioteca de Java	262
27.2. Per què calen dos conjunts de classes?	262
27.3. Tractament seqüencial o aleatori	263
27.4. Java IO: Streams	263
27.5. Java IO: Readers i Writers	264
28. Errors a la lectura/escriptura de fitxers	267
29. Lectura i escriptura de fitxers binaris d'accés seqüencial	269
29.1. Classe java.io.FileInputStream i FileOutputStream	269
29.2. Escriure un fitxer Byte a Byte	269
29.3. Sobreescriure el fitxer o afegir dades al final	271
29.4. Llegir un fitxer Byte a Byte	271
29.5. Mètodes de la classe InputStream	273
29.5.1. Un Exemple	274
29.5.2. Un altre exemple	275

29.6. Mètodes de la classe OutputStream	275
29.6.1. Un exemple	276
29.7. Exemple Xifrat del cèsar per bytes	277
30. Lectura i escriptura de fitxers de text a baix nivell	279
30.1. Caràcters i Unicode	279
30.2. Classe java.io.OutputStreamWriter	280
30.2.1. Determinar la codificació de caràcters	281
30.2.2. Tancar un OutputStreamWriter	281
30.3. Sobreescriure el fitxer o afegir dades al final	281
30.4. Classe java.io.InputStreamReader	282
30.4.1. Mètode read()	283
30.4.2. Final de fitxer	283
30.5. Selecció de la codificació de caràcters	284
30.5.1. Tancar un InputStreamReader	284
30.6. Classe java.io.FileReader i java.io.FileWriter	284
30.7. Sobreescriure el fitxer o afegir dades al final	285
31. Lectura i escriptura de fitxers de text a alt nivell	287
31.1. Classe Scanner	287
31.2. Exemple lectura d'un fitxer de text mitjançant la classe Scanner ..	288
31.3. Mètodes de la classe java.util.Scanner	290
31.4. Classe java.io.PrintWriter	291
31.5. Utilització d'un objecte PrintWriter	291
31.6. Mètodes de la classe java.io.PrintWriter	293
31.7. Exemple d'utilització de PrintWriter	294
31.8. Llegir un fitxer de text per paraules	295
31.9. Classe Scanner i la codificació de caràcters	298
31.10. Llegir un fitxer de text caràcter a caràcter	298
31.11. Classificar caràcters	298
31.12. Llegir un fitxer de text línia a línia	298
31.13. Escanejar una cadena	298
Bibliografia	301

24

Resultats d'aprenentatge i criteris d'avaluació

1. Dissenya, prova i documenta programes que realitzen diferents operacions sobre fitxers, documentant el programa i les proves realitzades.
 1. Reconeix els conceptes relacionats amb fitxers.
 2. Reconeix els diferents tipus de fitxers.
 3. Estableix i diferencia les operacions a realitzar sobre els fitxers en el llenguatge de programació emprat.
 4. Utilitza correctament diferents operacions sobre fitxers.
 5. Modula adequadament els programes que gestionen fitxers.
 6. Dissenya, prova i documenta programes simples que gestionen fitxers.

25

Continguts

1. Gestió de fitxers:

1. Concepte i tipus de fitxers.
2. Operacions sobre fitxers seqüencials i relatius.
3. Disseny de programes de gestió de fitxers.
4. Modulació de les operacions sobre fitxers.

26

Gestió de fitxers

26.1. Entrada / Sortida

Els processos d'entrada/sortida treballen llegint dades d'un **origen** i escrivint dades a un **destí**.

L'origen i el destí **no són necessàriament fitxers**, per exemple, el teclat opera com l'entrada estàndard encapsulada en Java per l'objecte **System.in** i la pantalla opera com la sortida estàndard, encapsulada en Java dins de l'objecte **System.out**.

Tant l'entrada com la sortida estàndard es poden redireccionar a d'altres orígens i a d'altres destins.

26.2. Paquets **java.io** i **java.nio**

Els paquets **java.io** i **java.nio** contenen les classes que gestionen l'entrada i la sortida. Les classes del **java.nio** gestionen la entrada i la sortida de dades amb **buffers** a diferència de les classes del paquet **java.io** que treballen directament amb **streams** o **fluxos**.

En general les classes del paquet **java.io** són més senzilles d'utilitzar i són amb les que treballarem en aquest apartat, no obstant les diferències principals entre els dos paquets són les següents:

java.io	java.nio
Treballa amb "Streams"	Treballa amb "buffers"
Les operacions E/S bloquegen	Les operacions E/S no bloquegen

java.io	java.nio
	Treballa amb Selectors

Treballar amb "Streams" vs treballar amb "buffers"

- Treballar amb "streams" significa que els bytes es llegeixen directament d'un flux de dades, la gestió posterior d'aquests bytes només depèn del programa, els bytes no s'emmagatzemaven a cap cache. Per tant, no és possible, d'entrada, moure's endavant i endarrere pel flux de bytes.
- Quan es treballa amb "buffers" les dades no es llegeixen directament del flux de dades sinó que es guarden en un "buffer" i es processen des d'allà. Això permet moure's endavant i endarrere pel "buffer" segons convingui. No obstant, cal comprovar si les dades necessàries estan o no al "buffer" cosa que en complica la gestió.

Operacions E/S que bloquegen vs operacions E/S que no bloquegen

- **java.io** bloqueja el fil d'execució actual, això vol dir que quan es realitza una operació de lectura o d'escriptura l'execució del codi s'interromp en espera de la lectura o escriptura de les dades.
- Les classes de **java.nio** no bloquegen el fil d'execució quan realitzen operacions de lectura o escriptura, per tant es poden demanar dades d'un canal, obtenir només el que estigui disponible en aquell moment i seguir treballant en alguna altra cosa mentre s'esperen noves dades.

Selectors

Els selectors de **java.nio** permeten a un sol fil d'execució monitorar més d'un canal E/S i seleccionar dinàmicament quins d'ells tenen dades per processar.

26.3. Conèixer el directori de treball

El concepte de **directori de treball** està relacionat amb els sistemes operatius, no és un concepte de Java.

Quan un procés comença, utilitza el seu directori de treball per resoldre les rutes relatives a fitxers, quan s'executa un programa Java, la màquina virtual Java s'executa com un procés i com a conseqüència té un directori de treball actual. El valor del directori de treball actual per una màquina virtual Java depèn de com s'ha executat la comanda **java**.

Conèixer quin és el directori de treball quan s'està programant amb fitxers és molt important ja que determina quin és el directori arrel per a les rutes relatives utilitzades en el programa.

Per obtenir el directori de treball actual cal llegir la propietat del sistema **user.dir**.

```
String workingDir = System.getProperty("user.dir");
```

També es pot especificar el directori de treball actual que utilitzarà la màquina virtual de Java al arrencar:

```
java -Duser.dir=C:\nou-working-dir la-meva-classe
```

26.4. "Paths" o rutes

Cada fitxer es troba dins d'un directori dins el sistema de fitxers, ens podem referir als fitxers a través d'una ruta.



Una **ruta**, amb anglès **path** és la forma general d'un nom de fitxer o carpeta, de manera que identifica la seva localització en el sistema de fitxers.

Les rutes poden ser de dos tipus:

ruta absoluta d'un fitxer

És aquella que es refereix a un element destinació a partir de la carpeta arrel d'un sistema de fitxers. En aquesta s'enlacen, una per una, totes les carpetes que hi ha entre la carpeta arrel fins a la carpeta que conté l'element destinació.

Per exemple, **c:\javaprojects\Exercici.java** o bé **/etc/network/interfaces**.

ruta relativa d'un fitxer

És aquella que es considera que parteix des del **directori de treball** de l'aplicació. Aquesta carpeta pot ser diferent cada cop que s'executa el programa i depèn de la manera com s'ha dut a terme aquesta execució.

Per exemple, si el directori de treball actual és **c:\javaprojects**, **Exercici.java** és la ruta relativa al fitxer **c:\javaprojects\Exercici.java**.



Cal tenir sempre present si el sistema operatiu distingeix entre majúscules i minúscules o no.



El format de la cadena de text que conforma la ruta pot ser diferent segons el sistema operatiu sobre el qual s'executa l'aplicació.

Per exemple, el sistema operatiu Windows inicia les rutes per un nom d'unitat (C:, D:, etc.), mentre que els sistemes operatius basats en Unix comencen directament amb una barra ("/").

A més a més, els diferents sistemes operatius usen diferents separadors dins les rutes.

Per exemple, els sistemes Unix usen la barra ("/") mentre que el Windows la contrabarra ("\").

Veure l'apèndix [Representacions de rutes per sistema operatiu i shell](#)

26.5. Tractament d'errors en l'accés a fitxers

Una dificultat afegida quan es treballa amb fitxers en Java és el tractament d'errors o d'**excepcions** si utilitzem la nomeclatura pròpia de Java.

El problema de fons és que els fitxers són recursos externs a Java i poden estar o no disponibles o accessibles independentment de la gestió que fem com a programadors. Per exemple, és possible que s'intenti accedir a un fitxer inexistent, o crear un fitxer a una ubicació en la que no tenim permisos d'escriptura.

EL punt és que Java ens obliga a gestionar aquestes situacions, que Java anomena excepcions, el compilador insisteix en que especifiquem què és el que ha de fer el programa quan es aquestes es produueixen, si no li especifiquem el programa no es deixa compilar.

El tractament d'aquestes excepcions és un tema que es tracta en futures UF però per seguir endavant hem de prendre una solució de compromís, ja que Java no ens deixa ignorar aquestes situacions.

Per sortir del pas indicarem al compilador que si no es troba el fitxer indicat es dispari l'excepció en forma d'error d'execució, s'interrompi l'execució i es mostri la informació de l'excepció per la sortida estàndard i que actuï de la mateixa manera si es produeix una excepció quan s'intenta la lectura d'un valor.

La solució temporal que seguirem serà la següent:

Els mètodes que manipulin classes d'accés a fitxers, és a dir que continguin instàncies de File, Streams, Readers o Writers els decorarem amb la següent clàusula **throws IOException** per indicar que és possible que aquests mètodes generin excepcions en la manipulació dels fitxers que gestionen.

Per exemple:

```
public static void main(String[] args) throws IOException {  
}
```



Cal tenir molt present que la solució presa **no** és en cap cas la indicada en una aplicació real, on en cas de trobar problemes amb l'accés als fitxers **intentariem donar la opció al usuari de subsanar el problema, permetent-li canviar de ruta o de nom el fitxer per exemple.**

26.6. La classe File

La classe **File** és una **representació abstracte dels noms de les rutes a carpetes i fitxers**. Conté mètodes per **obtenir les propietats** d'un fitxer o d'un directori i per **renombrar i eliminar** un fitxer o un directori.

**File(pathname: String)**

Crea un objecte File per la ruta especificada. *pathname* pot ser un fitxer o un directori.

File(parent: String, child: String)

Crea un objecte File per *child* dins del directori parent. *child* pot ser un fitxer o un directori.

File(parent: File, child: String)

Crea un objecte File per *child* dins del directori parent. *parent* és un objecte File.

exists(): boolean

Retorna true si el fitxer o directori representat per l'objecte File existeix. El mètode **exists** comprova si el path assignat a un objecte File existeix.

canRead: boolean

Retorna true si el fitxer especificat existeix i té permisos de lectura per l'aplicació.

canWrite: boolean

Retorna true si el fitxer especificat d'escriptura per l'aplicació.

createNewFile(): boolean

Crea un fitxer nou, buit si el nom especificat a l'objecte File no existeix.

isDirectory

Retorna true si el fitxer indicat per la ruta és un directori.

isFile()

Retorna true si el fitxer indicat per la ruta és un fitxer.

isAbsolute()

Retorna true si el fitxer indicat per la ruta és una ruta absoluta.

isHidden()

Retorna true si el fitxer indicat per la ruta és està marcat com a ocult.

getAbsolutePath(): String

Retorna la ruta sencera del fitxer o directori representat per l'objecte File.

getCanonicalPath(): String

Retorna el mateix que el mètode **getAbsolutePath** però elimina tota la informació redundant de la ruta.

Elimina els . i .., resol els links simbòlics, en Linux, i posa la lletra de la unitat en majúscula, en Windows.

getName(): String

Retorna el nom del fitxer o del directori indicat pel path.

getPath(): String

Retorna la ruta encapsulada al objecte File en una cadena.

getParent(): String

Retorna la ruta del directori pare del fitxer. REtorna null en cas que el fitxer no tingui un directori pare.

lastModified(): long

Retorna el temps de l'última modificació del fitxer.

length(): long

Retorna la mida del fitxer en bytes.

listfiles(): File[]

Retorna una matriu de rutes indicant els fitxers del directori indicat pel path.

listRoots():File[]

Retorna una matriu indicant els diferents directoris arrel del sistema de fitxers.

delete(): boolean

Elimina el fitxer o directori representat per l'objecte File. Retorna **true** si l'eliminació té èxit.

renameTo(dest: File): boolean

Renombra a **dest** el fitxer o directori representat per l'objecte File. Retorna **true** si l'operació té èxit.

setReadonly(): boolean

Marca el fitxer o el directori com a només lectura.

setReadable(boolean readable): boolean

Proporciona permisos de lectura pel propietari del fitxer. Retorna **true** si l'operació ha tingut èxit.

setWritable(boolean writable): boolean

Proporciona permisos d'escriptura pel propietari del fitxer. Retorna **true** si l'operació ha tingut èxit.

setExecutable(boolean executable): boolean

Proporciona permisos d'execució pel propietari del fitxer. Retorna **true** si l'operació ha tingut èxit.

mkdir(): boolean

Crea el directori representat per l'objecte File. Retorna **true** si el directori es crea amb èxit.

mkdirs(): boolean

Funciona igual que **mkdir()** excepte que crea el directori junt amb els directoris pare si aquests no existeixen.

26.7. Creació d'un objecte File

Es pot crear un objecte File a partir d'una ruta relativa o absoluta.

Per exemple:

```
File fitxerTest = new File("fitxerTest.txt");
```



El caràcter (*/*) és el separador de directoris en Java, el mateix que Unix. Java és capaç de traduir una ruta amb aquest separador als altres sistemes operatius de forma automàtica.



Crear un objecte File **no crea cap fitxer a l'ordinador**. Es pot crear una instància per qualsevol nom de fitxer independent de si existeix o no.

El mètode **exists()** de la classe File permet veure si el fitxer existeix o no.

File(pathname: String)

Crea un objecte File per la ruta especificada, *pathname* pot ser un fitxer o un directori.

File(parent: String, child: String)

Crea un objecte File per child dins del directori parent. child pot ser un fitxer o un directori.

File(parent: File, child: String)

Crea un objecte File per child dins del directori parent. parent és un objecte File.

26.8. Comprovar si el fitxer existeix

El mètode **exists()** de la classe **File** permet determinar si la ruta encapsulada a l'objecte **File** existeix o no.

exists(): boolean

Retorna **true** si el fitxer o directori representat per l'objecte File existeix.

```
File testFile = new File("dummy.txt");

boolean fileExists = testFile.exists();
if (fileExists) {
    System.out.println("testFile.txt existeix.");
} else {
    System.out.println("textFile.txt no existeix.");
}
```

26.9. Comprovar si el fitxer és un directori

Un objecte de tipus **File** pot ser tant un directori, com un fitxer. Per determinar què és es poden utilitzar els mètodes **isDirectory()** i **isFile()**.

isDirectory

Retorna true si el fitxer indicat per la ruta és un directori.

isFile()

Retorna true si el fitxer indicat per la ruta és un fitxer.

26.10. Ruta absoluta i ruta canònica

Una **ruta absoluta** identifica un fitxer dins un sistema de fitxers. El problema és que es poden construir moltes rutes absolutes que fan referència al mateix fitxer. I això pot ser un problema, en particular si es vol determinar si dues rutes són la mateixa.

Per exemple, les següents rutes fan referència al mateix fitxer:

```
C:/test/testFile.txt
C:/test/temporals/../testFile.txt
```

```
C:/test/temporals/../temporals/../testFile.txt
```

Una **ruta canònica** és la ruta absoluta més simple que fa referència a un fitxer.

Els mètodes **getAbsolutePath()** i **getCanonicalPath()** de la classe **File** retornen la ruta absoluta i la ruta canònica de la ruta encapsulada dins del objecte **File**.

getAbsolutePath(): String

Retorna la ruta sencera del fitxer o directori representat per l'objecte File.

getCanonicalPath(): String

Retorna el mateix que el mètode getAbsolutePath però elimina tota la informació redundant de la ruta.

Elimina els . i .., resol els **links simbòlics**, en Linux, i posa la **lletre de la unitat en majúscula**, en Windows.

Per exemple:

```
import java.io.File;
import java.io.IOException;

public class FileDemo {

    public static void main(String[] args) throws IOException {
        File f1 = new File("/test/../test/fitxer.txt");

        System.out.println("Name: " + f1.getName());
        System.out.println("Path: " + f1.getPath());
        System.out.println("Absolute Path: " + f1.getAbsolutePath());
        System.out.println("Canonical Path: " + f1.getCanonicalPath());
        System.out.println("Parent: " + f1.getParent());
        System.out.println(f1.exists() ? "exists" : "does not exist");
        System.out.println(f1.canWrite() ? "is writeable" : "is not
writeable");
        System.out.println(f1.canRead() ? "is readable" : "is not
readable");
        System.out.println("is " + (f1.isDirectory() ? "" : "not" + " a
directory"));
        System.out.println(f1.isFile() ? "is normal file" : "might be a
named pipe");
        System.out.println(f1.isAbsolute() ? "is absolute" : "is not
absolute");
    }
}
```

```

        System.out.println("File last modified: " + f1.lastModified());
        System.out.println("File size: " + f1.length() + " Bytes");
    }

}

```

```

Name: fitxer.txt
Path: \test..\test\fitxer.txt
Absolute Path: C:\test..\test\fitxer.txt
Canonical Path: C:\test\fitxer.txt
Parent: \test..\test
does not exist
is not writeable
is not readable
is not a directory
might be a named pipe
is not absolute
File last modified: 0
File size: 0 Bytes

```



No és recomanable treballar amb rutes absolutes dins dels programes, les rutes absolutes són susceptibles dels canvis de sistema operatiu, dels canvis de directori de treball i dels canvis d'ubicació del programa.

Si es treballa amb rutes absolutes, aquestes haurien d'emmagatzemar fora del programa, en un fitxer de configuració, passades com a paràmetres d'inici, etc...

26.11. Crear, renombrar i eliminar fitxers

createNewFile(): boolean

El mètode **createNewFile()** crea un fitxer nou, buit, si el nom especificat a l'objecte File no existeix. El mètode llança una excepció de tipus **IOException** si es produeix un error d'entrada/sortida.

Per exemple:

```

File dummyFile = new File("dummy.txt");
boolean fileCreated = dummyFile.createNewFile();

```

delete(): boolean

El mètode **delete()** elimina el fitxer o directori especificat. Els directoris han d'estar buits per a poder-se eliminar. El mètode retorna **true** si el fitxer o directori s'ha eliminat correctament, en cas contrari retorna **false**.

Per exemple:

```
File dummyFile = new File("dummy.txt");
dummyFile.delete();
```

mkdir(): boolean

Crea el directori representat per l'objecte File. Retorna **true** si el directori es crea amb èxit.

mkdirs(): boolean

Els mètodes **mkdir()** i **mkdirs()** creen nous directoris. El mètode **mkdir()** crea un nou directori si el directori pare especificat a la ruta existeix. El mètode **mkdirs()** crea el directori si el directori pare no existeix.

Per exemple:

```
File newDir = new File("C:\\users\\home");
newDir.mkdir(); // Crea el directori home si existeix c:\\users
newDir.mkdirs(); // Crea el directori c:\\users\\home si c:\\users no
                 existeix
```

renameTo(dest: File): boolean

El mètode **renameTo()** permet canviar el nom d'un fitxer. Aquest mètode retorna **true** si el fitxer s'ha eliminat correctament, en cas contrari retorna **false**.

Per exemple:

```
File oldFile = new File("old_dummy.txt");
File newFile = new File("new_dummy.txt");
boolean fileRenamed = oldFile.renameTo(newFile);
if (fileRenamed) {
    System.out.println(oldFile + " renamed to " + newFile);
} else {
```

```
        System.out.println("Renaming " + oldFile + " to " + newFile + "  
failed.");  
    }
```

26.12. Treballar amb els atributs dels fitxers

La classe **File** conté alguns mètodes que permeten manipular els atributs dels fitxers de forma limitada.

setReadonly(): boolean

Marca el fitxer o el directori com a només lectura.

setReadable(boolean readable): boolean

Proporciona permisos de lectura pel propietari del fitxer. Retorna true si l'operació ha tingut èxit.

setWritable(boolean writable): boolean

Proporciona permisos d'escriptura pel propietari del fitxer. Retorna true si l'operació ha tingut èxit.

setExecutable(boolean executable): boolean

Proporciona permisos d'execució pel propietari del fitxer. Retorna true si l'operació ha tingut èxit.

Per exemple:

```
import java.io.File;  
  
public class FileTest {  
  
    public static void main(String[] args) {  
  
        File f = null;  
        boolean bool = false;  
  
        // Creem un objecte File  
        f = new File("C:/test.txt");  
        if (f.exists()) {  
            // Establim permís de lectura  
            bool = f.setReadable(true);  
        }  
    }  
}
```

```
System.out.println("setReadable() estableert?: " + bool);

// Comprovem si el fitxer té permisos de lectura
bool = f.canRead();

System.out.print("El fitxer té permisos de lectura?: " +
bool);
}
```

26.13. Copiar un fitxer

La classe **File** no proporciona cap mètode que permeti copiar un fitxer.

Per copiar un fitxer cal crear-ne un de nou, llegir el contingut del fitxer original i escriure'l al nou fitxer.

Com alternativa és pot utilitzar el mètode **copy** de la classe **java.nio.Files** que proporciona varies sobrecàrregues.

26.14. Saber la mida d'un fitxer

El mètode **length()** retorna la mida d'un fitxer en bytes.

length(): long

Retorna la mida del fitxer en bytes.

26.15. Mostrar tots els directoris arrel

Es pot obtenir una llista de tots els directoris arrel d'un sistema de fitxers utilitzant el mètode **listRoots()**.

Els directoris arrel depenen del sistema operatiu, en Windows existeix un directori arrel per a cada unitat de disc (c:\, d:\, etc...) en Linux hi ha un únic directori arrel representant per (\)

```
import java.io.File;

public class RootList {
```

```

public static void main(String[] args) {
    File[] roots = File.listRoots();
    System.out.println("Directoris arrel:");
    for (File f : roots) {
        System.out.println(f.getPath());
    }
}
}

```

```

Directoris arrel:
C:\ 
D:\ 
G:\ 
Y:\ 
Z:\ 

```

26.16. Mostrar tots els fitxers i directoris d'un directori

La classe **File** proporciona mètodes per veure el contingut d'un directori.

listfiles(): File[]

Retorna una matriu de rutes indicant els fitxers del directori indicat pel path.

list(): String[]

Retorna una matriu de cadenes indicant els **noms** dels fitxers del directori indicat pel path.

```

import java.io.File;

public class FileTest {

    public static void main(String[] args) {
        File directoriArrel = new File(System.getProperty("user.dir"));
        File[] fitxers = directoriArrel.listFiles();
        for (int i = 0; i < fitxers.length; i++) {
            if (fitxers[i].isDirectory()) {
                System.out.print("[D] ");
            } else if (fitxers[i].isFile()) {
                System.out.print("[F] ");
            } else {
                System.out.print("[?] ");
            }
        }
    }
}

```

```
        System.out.println(fitxers[i].getName());  
    }  
}  
}
```

```
[F] build.xml  
[F] f.txt  
[F] manifest.mf  
[D] nbproject  
[D] src  
[D] test
```


27

Lectura i escriptura de fitxers

Essencialment hi ha dues maneres d'**interpretar** dades emmagatzemades, en **format text** o en **format binari**.

- En **format text** les dades es s'interpreten com una seqüència de caràcters. A més, aquesta interpretació és susceptible de la codificació de caràcters emprada.
- En **format binari** les dades s'interpreten com a una seqüència de **bytes**.

Per exemple:

- en format text i codificació ASCII, la cadena **àabc** s'emmagatzemaria com la seqüència de bits,

```
11100000 01100001 01100010 01100011
```

- però la mateixa cadena en codificació UTF-8, s'emmagatzemaria amb una seqüència de bits diferent,

```
11000011 10100000 01100001 01100010 01100011
```

Si llegim les dades de l'exemple com a text obtindrem en ambdós casos la mateixa cadena, **àabc**. En canvi si llegim les dades en format binari obtindrem diferents seqüències de bits.

27.1. Classes de la biblioteca de Java

La biblioteca de Java proporciona dos conjunts de classes per gestionar la entrada i la sortida.

Classes **xxxStream**

La família de classes **xxxStream** gestionen les dades d'un origen de dades de forma **binària**, és a dir, byte a byte.

Classes **xxxReader** i **xxxWriter**

La família de classes **xxxWriter** i **xxReader** gestionen les dades en format text, és a dir, caràcter a caràcter.

27.2. Per què calen dos conjunts de classes?

Tot i que els caràcters estan formats per bytes existeixen variacions respecte com es representa cada caràcter. Per exemple, el caràcter 'é' es codifica com un únic bit de valor 223 en la codificació ISO-8859-1, típicament utilitzada a Nord Amèrica i a Europa occidental. En canvi en la codificació UTF-8 el caràcter 'é' es codifica amb dos bytes, 195 i 169, i en la codificació UTF-16 es codificantaria com a 0 223.

Les classes **Reader** i **Writer** tenen la responsabilitat de convertir entre bytes i caràcters, tenint en compte les diferents codificacions de caràcters involucrades.

Per defecte utilitzen la configuració de caràcters especificada pel sistema operatiu que executa el programa, no obstant, es pot especificar una codificació diferent a l'hora de construir l'objecte Reader o Writer. Per exemple:

```
Scanner in = new Scanner(input, "UTF-8");
PrintWriter out = new PrintWriter(output, "UTF-8");
```



No hi ha manera de determinar automàticament la codificació de caràcters utilitzada en un text particular. Cal conèixer quina codificació es va utilitzar en el moment de la escriptura del text i adaptar les classes de lectura a aquesta codificació.

27.3. Tractament seqüencial o aleatori

A part de tenir classes per tractar fitxers en base al seu contingut també es proporcionen diferents classes per tractar fitxers en base a la manera d'accendir-hi.

Fitxers d'accés seqüencial

Com que els valors d'un fitxer es solen emmagatzemar en forma de seqüència, un rere l'altre, la manera més habitual de tractar fitxers és seqüencialment.

S'anomena **accés seqüencial** al tractament d'un conjunt d'elements de manera que només és possible accedir-hi d'acord al seu l'ordre d'aparició i per tant, per a poder tractar un element, cal haver tractat tots els elements anteriors.

Fitxers d'accés aleatori o accés directe

L'accés aleatori fa referència a la habilitat d'accendir a una dada en unes coordenades donades dins d'un conjunt d'elements adreçables.

27.4. Java IO: Streams

Els següents diagrames mostren part de la jerarquia de classes que treballen amb fluxos de dades, és a dir, aquestes classes **accedeixen a les dades com a un a seqüència de bytes**. En general les dades s'extrauran o s'enviaran a connexions de xarxa, fitxers o dispositius externs.

En funció de les nostres necessitats s'escolllirà una de les classes o una altra, per exemple, la solució més senzilla per treballar amb orígens i destins de tipus fitxer serà escolllir les classes **FileInputStream** i **FileOutputStream**.

Es mostren dues jerarquies de classes, en primer lloc les classes que s'utilitzen per **llegir** dades seqüencialment i en format binari i en segon lloc les classes que s'utilitzen per **escriure** dades seqüencialment en format binari.

Quan parlem de **jerarquia** fem referència a que les classes situades més avall, a la cua de les fletxes, són **especialitzacions** de les situades més amunt, per exemple, en el diagrama següent estem dient que un **PrintStream** és en particular un **FilterOutputStream** i aquest és en particular un **OutputStream**.

Com a conseqüència un **PrintStream** tindrà tots els mètodes que té un **FilterOutputStream** i possiblement més, i un **FilterOutputStream** tindrà tots els mètodes que té un **OutputStream** i possiblement més.

En general, aquestes classes no mantenen un index per llegir o escriure les dades i en general no permet la lectura o escriptura endavant o endarrere del stream.

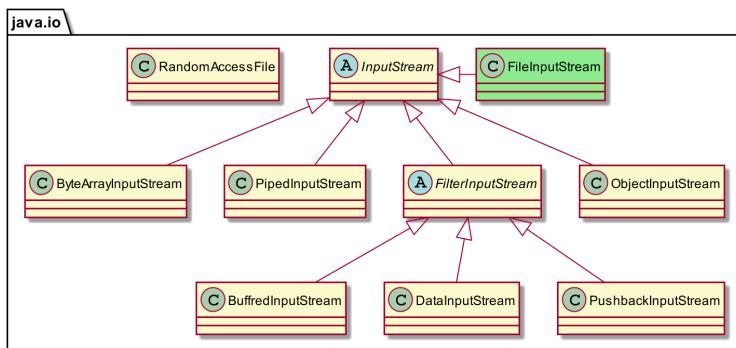


Figura 27.1. Classes habituals de lectura de "streams" binaris

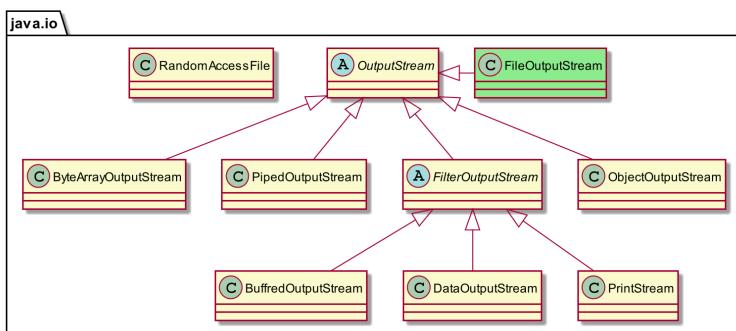


Figura 27.2. Classes habituals d'escriptura de "streams" binaris

27.5. Java IO: Readers i Writers

Els següents diagrames mostren part de la jerarquia de classes que treballen amb "readers" i "writers", és a dir, aquestes classes **accedeixen a les dades com a un a seqüència de caràcters**.

En funció de les nostres necessitats s'escolllirà una de les classes o una altra, per exemple, la solució més senzilla per treballar amb orígens i destins de tipus fitxer serà escollir les classes **FileReader** i **FileWriter**.

Es mostren dues jerarquies de classes, en primer lloc les classes que s'utilitzen per **llegir** dades seqüencialment i en format caràcter i en segon lloc les classes que s'utilitzen per **escriure** dades seqüencialment en format caràcter..

Les classe **java.io.Reader** i la classe **java.io.Writer** funcionen de forma similar a les classes **InputStream** i **OutputStream** amb la diferència que les primeres són basades en **caràcters** i es segones són basades en bytes. Estan pensades per **treballar amb text**.

En general, aquestes classes no mantenen un index per llegir o escriure les dades i en general no permet la lectura o escriptura endavant o endarrere del stream.

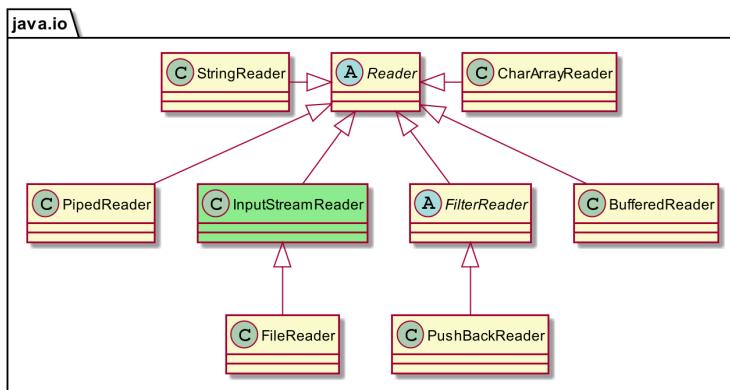


Figura 27.3. Part de la jerarquia de classes orientades a la lectura de fitxers de text

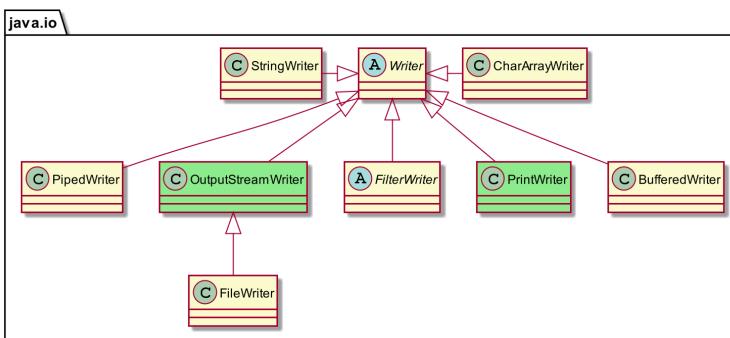


Figura 27.4. Part de la jerarquia de classes orientades a l'escriptura de fitxers de text

28

Errors a la lectura/escriptura de fitxers

Els errors que es produeixen durant la lectura/escriptura de fitxers tenen una tractament especial.

És una decisió de disseny de Java: Java considera que els errors **potencials** d'entrada i sortida han de ser tractats **obligatòriament** pel programador.

Això vol dir, per exemple, que com a programadors hem d'acceptar **explícitament** que quan accedim a un fitxer es pot generar una excepció de "fitxer inexistent".

A la pràctica tenim dues opcions:

La primera opció consisteix en "**gestionar**" l'excepció i reconduir l'execució del programa encara que s'ha produït una situació d'error. Per exemple, en el cas que el fitxer no existeixi, gestionem l'excepció tornant a demanar a l'usuari la ruta d'un nou fitxer. **Aquesta opció no la tractarem en aquesta UF**

La segona opció consisteix en notificar **explícitament** a Java que estem disposats a assumir la interrupció del programa quan es produeix un a excepció d'entrada sortida.

Per fer-ho cal "decorar" la declaració del mètode on es produeix l'excepció amb les clàusules "**throws IOException**:

Per exemple:

```
public static void main(String[] args) throws IOException {
```

```
// Accedim a fitxers  
}
```

Sembla embolicat però no ho és ja que si no recordem fer-ho **el propi netbeans es queixa** i per tan no **ens passa per alt**.

29

Lectura i escriptura de fitxers binaris d'accés seqüencial

En primer lloc anem a donar un cop d'ull a algunes de les classes que permeten treballar amb Fitxers binaris de forma seqüencial.

29.1. Classe `java.io.FileInputStream` i `FileOutputStream`

Per llegir dades binàries d'un fitxer en un disc cal crear un objecte `FileInputStream`.

```
File f = new File("input.bin");
FileInputStream inputStream = new FileInputStream(f);
```

De forma similar, si el que volem es escriure dades en un fitxer binari en un disc, utilitzem la classe `FileOutputStream`

```
File f = new File("output.bin");
FileOutputStream outputStream = new FileOutputStream(f);
```

29.2. Escriure un fitxer Byte a Byte

Per escriure un fitxer Byte a Byte utilitzarem `FileOutputStream` i els mètodes següents:

Mètode `write`

La classe `FileOutputStream` disposa del mètode `write` que permet escriure un sol byte al fitxer indicat. La documentació del mètode `write` indica que cal

passar-li un paràmetre de tipus **int**, en realitat el mètode només agafa els els ultims 8 bits de l'enter passat per paràmetre. A la pràctica, doncs, li estarem passant un **byte** que serà la dada que emmagatzemarà.



Escriure un byte amb **FileOutputStream** crea automàticament el fitxer indicat en cas que aquest no existeixi.

Mètode close

Un cop finalitzada l'escriptura de les dades caldrà cridar al mètode **close()** per garantir que s'alliberen els recursos associats.

Per exemple:

```
import java.io.File;
import java.io.FileOutputStream;

import java.io.IOException;

public class TestFileIO {

    public static void main(String[] args) throws IOException { ❶
        File f = new File("test.bin"); ❷
        FileOutputStream fos = new FileOutputStream(f); ❸
        byte valor = 65; ❹
        fos.write(valor); ❺
        valor = 66;
        fos.write(valor); ❻

        fos.close(); ❻
    }
}
```

- ❶ Indiquem que el mètode pot generar una excepció de tipus **IOException**
- ❷ Indiquem el fitxer destí. Aquest fitxer no cal que existeixi, en aquest cas es crearà automàticament amb la primera operació d'escriptura.
- ❸ Creem un **FileOutputStream** passant-li el fitxer destí.
- ❹ Preparam el byte que volem escriure.

- ⑤ Escrivim el byte amb l'ajuda del mètode **write**
- ⑥ Seguim escrivint bytes
- ⑦ Al acabar les operacions d'escriptura tanquem el **FileOutputStream** amb l'ajuda del mètode **close()**

29.3. Sobreescrivir el fitxer o afegir dades al final

Per defecte **FileOutputStream** elimina **tot** el contingut del fitxer abans de començar a escriure.

Tenim la opció d'obrir el fitxer **per afegir dades al final i mantenir tot el contingut actual**. Per fer-ho cal afegir un paràmetre **true** durant la creació del **FileOutputStream** tal i com es mostra al següent exemple:

```
import java.io.File;
import java.io.FileOutputStream;

import java.io.IOException;

public class TestFileIO {

    public static void main(String[] args) throws IOException {
        File f = new File("test.bin");
        FileOutputStream fos = new FileOutputStream(f, true); ❶
        byte valor = 65;
        fos.write(valor);
        valor = 66;
        fos.write(valor);

        fos.close();
    }
}
```

- ❶ El paràmetre booleà a **true** habilita l'addició de dades al final del fitxer.

29.4. Llegir un fitxer Byte a Byte

Per llegir un fitxer Byte a Byte utilitzarem **FileInputStream** i els mètodes següents:

Mètode read

La classe **FileInputStream** té un mètode, **read**, que permet llegir un **byte** cada cop des d'un fitxer de disc.

El mètode **read** retorna un **int**, no un byte, d'aquesta manera pot indicar que s'ha arribat al final del fitxer retornant un **-1**, en cas contrari retorna un valor entre -128 i 127. (Recordeu que els bytes en Java tenen signe!!)

Mètode close

Un cop finalitzada la lectura de les dades caldrà cridar al mètode **close()** per garantir que s'alliberen els recursos associats.

Per exemple:

```
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;

public class TestFileIO {

    public static void main(String[] args) throws IOException { ❶
        File f = new File("test.bin"); ❷
        FileInputStream fis = new FileInputStream(f); ❸
        int nextInt = fis.read(); ❹
        while (nextInt != -1) { ❺
            byte nextByte = (byte) nextInt; ❻
            System.out.println(nextByte); ❼
            nextInt = fis.read(); ❽
        }

        fis.close(); ❾
    }
}
```

- ❶ Indiquem que el mètode pot generar una excepció de tipus **IOException**
- ❷ Indiquem el fitxer destí.
- ❸ Creem un **FileInputStream**_ passant-li el fitxer destí.
- ❹ Llegim el primer enter. (Que en realitat és -1 (int) o un byte)

- ⑤ Si el enter llegit és diferent de -1 vol dir que no hem arribat a final de fitxer
 - ⑥ En aquest punt sabem que el enter llegit és efectivament un byte i per tant podem fer un cast
 - ⑦ Gestionem el byte llegit de la manera que necessitem, en l'exemple el mostrem per pantalla
 - ⑧ Llegim el següent byte del fitxer.
 - ⑨ Al acabar el procés tanquem el **FileInputStream** per alliberar els recursos.
- ====

Aquests són els únics mètodes que proporcionen les classes **FileInputStream** i **FileOutputStream**, si es volen llegir números, cadenes o objectes caldrà utilitzar alguna de les classes amb capacitat d'agrupar bytes individuals en estructures més complexes.

29.5. Mètodes de la classe FileInputStream

C	FileInputStream
<ul style="list-style-type: none"> ● <code>read(): int</code> ● <code>read(b: byte[]): int</code> ● <code>read(b: byte[], off: int, len: int): int</code> ● <code>available(): int</code> ● <code>close(): void</code> ● <code>skip(n: long): long</code> ● <code>markSupported(): boolean</code> ● <code>mark(readlimit: int): void</code> ● <code>reset(): void</code> 	

read(): int

Llegeix el següent byte del flux d'entrada. El valor retornat és un int entre 0 i 255. Si s'ha arribat al final del flux es retorna -1.

read(b: byte[]): int

Llegeix fins a b.length del flux d'entrada i retorna la quantitat de bytes llegits. Retorna -1 al final del flux.

read(b: byte[], off: int, len: int): int

Llegeix bytes del flux d'entrada i els emmagatzema a b[off], b[off+1], ..., b[off +len-1]. Retorna la quantitat de bytes llegits. Retorna -1 al final del flux.

available(): int

Retorna la quantitat de bytes estimats que es poden llegir al flux d'entrada.

close(): void

Tanca el flux d'entrada i allibera els recursos associats.

skip(n: long): long

Salta i descarta **n** bytes de dades. Es retorna la quantitat de bytes descartats.

markSupported(): boolean

Determina si el flux suporta els mètodes **mark** i **reset**. En el cas de **FileInputStream** els mètodes anteriors **no** són suportats.

mark(readlimit: int): void

Marca la posició actual en el flux d'entrada.

reset(): void

Reposiciona l'apuntador del flux a la última posició definida pel mètode **mark**.

29.5.1. Un Exemple

El següent exemple mostra el fitxer especificat byte a byte, si el fitxer és de text i està codificat en ASCII es mostraran els codi ASCII de les lletres.

```
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;

public class StreamDemo {

    public static void main(String[] args) throws IOException {

        File f = new File("c:\\\\data\\\\text.txt");
        FileInputStream inputstream = new FileInputStream(f);

        int data = inputstream.read();
        while (data != -1) {
            ferAlgunaCosaAmbElByte(data);
            data = inputstream.read();
        }
        inputstream.close();
    }
}
```

```

public static void ferAlgunaCosaAmbElByte(int b) {
    System.out.println((byte) b);
}
}

```

29.5.2. Un altre exemple

El següent exemple mostra el fitxer especificat en hexadecimal.

```

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;

public class StreamsDemo {

    public static void main(String[] args) throws IOException {
        File f = new File("manifest.mf");
        FileInputStream is = new FileInputStream(f);
        int nextByte;
        String mask = "00000000";
        do {
            nextByte = is.read();
            System.out.printf("%2hh ", nextByte);
        } while (nextByte != -1);
    }
}

```

29.6. Mètodes de la classe OutputStream

	OutputStream
● write(int b): void	
● write(b: byte[], off: int, len: int): void	
● write(b: byte[]): void	
● close(): void	
● flush(): void	

write(int b): void

Escriu el byte especificat al flux de sortida. El paràmetre **b** és de tipus **int**.

write(b: byte[], off: int, len: int): void

Escriu b[off], b[off+1], ..., b[off+len-1] al flux de sortida.

write(b: byte[]): void

Escriu tots els bytes de la matriu al flux de sortida.

close(): void

Tanca el flux de dades i n'allibera tots els recursos associats.

flush(): void

Buida el flux de sortida i força l'escriptura de tots els bytes del buffer.

29.6.1. Un exemple

El següent exemple copia d'un fitxer.

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class StreamDemo {

    public static void main(String[] args) throws IOException {

        File origen = new File("imatge.png");
        File copia = new File("imatge_copia.png");
        FileInputStream in = new FileInputStream(origen);
        FileOutputStream out = new FileOutputStream(copia);

        int b = in.read();
        while (b != -1) {
            out.write(b);
            b = in.read();
        }

        in.close();
        out.close();
        System.out.println("Imatge copiada amb èxit");
    }
}
```

29.7. Exemple Xifrat del cèsar per bytes

El següent programa encripta un fitxer agafant cadascun dels bytes i sumant-els-hi **n** tenint en compte que si el valor resultant sobrepassa 255 es dona la volta començant altre cop per 0.

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Scanner;

/**
 * Aquest programa encripta un fitxer utilitzant el xifrat del cèsar.
 */
public class CaesarEncryptor {

    public static void main(String[] args) throws IOException {
        Scanner in = new Scanner(System.in);

        System.out.print("Fitxer d'entrada: ");
        String inFile = in.nextLine();
        System.out.print("Fitxer de sortida: ");
        String outFile = in.nextLine();
        System.out.print("Clau d'encriptació: ");
        int clau = in.nextInt();

        FileInputStream inStream = new FileInputStream(inFile);
        FileOutputStream outStream = new FileOutputStream(outFile);

        encriptarFlux(clau, inStream, outStream);
        inStream.close();
        outStream.close();
    }

    /**
     * Encripta el contingut d'un flux.
     *
     * @param clau clau d'encriptació
     * @param in flux d'entrada
     * @param out flux de sortida
     */
    public static void encriptarFlux(int clau, FileInputStream in,
        FileOutputStream out) throws IOException {
```

```
int next = in.read();
while (next != -1) {
    int encrypted = encriptar(next, clau);
    out.write(encrypted);
    next = in.read();
}
}

/**
 * Encripta un valor.
 *
 * @param b el byte a encriptar (entre 0 i 255)
 * @param clau la quantitat sumada al byte
 * @return byte encriptat
 */
public static int encriptar(int b, int clau) {
    return (b + clau) % 256;
}
```

30

Lectura i escriptura de fitxers de text a baix nivell

Similarment al que hem fet amb l'accés seqüencial byte a byte podem treballar accedint seqüencialment a un **fitxer de text** char a char.

30.1. Caràcters i Unicode

A dia d'avui, moltes aplicacions utilitzen UTF (UTF-8 o UTF-16) per emmagatzemar dades en format text.

Per exemple, en UTF-8 els caràcters ocupen **un o dos bytes**, en UTF-16 els caràcters ocupen **2 bytes**.

Per tant, l'hora de llegir o escriure text és molt complicat fer-ho byte a byte.

Per solucionar-ho tenim la família de classes **Reader** i la família de classes **Writer**. Les classes **Reader** i **Writer** són capaces de descodificar bytes en caràcters, per fer-ho, **cal informar de la codificació de caràcters a utilitzar** just en el moment de la seva creació.

30.2. Classe java.io.OutputStreamWriter

C	OutputStreamWriter
<ul style="list-style-type: none"> ● OutputStreamWriter(out: OutputStream) ● OutputStreamWriter(out: OutputStream, cs: Charset) ● OutputStreamWriter(out: OutputStream, enc: CharsetEncoder) ● OutputStreamWriter(out: OutputStream, charsetName: String) ● close(): void ● flush(): void ● getEncoding(): String ● write(cbuf: char[], off: int, len: int): void ● write(c: int): void ● write(str: String, off: int, len: int): void 	

La classe **OutputStreamWriter** està pensada per encapsular un **OutputStream**, d'aquesta manera es pot convertir un stream d'escriptura basat en bytes a un **Writer** basat en caràcters.

Per tant, l'objectiu d'aquesta classe és **adaptar un objecte capaç d'escriure a un flux byte a byte en un objecte que es capaç d'escriure sobre el mateix flux de dades però caràcter a caràcter**.

Per exemple:

```
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStreamWriter;

public class FileTest {

    public static void main(String[] args) throws IOException {
        File f = new File("/data/output.txt"); ①
        FileOutputStream outputStream = new FileOutputStream(f); ②
        OutputStreamWriter outputStreamWriter = new
        OutputStreamWriter(outputStream, "utf-8"); ③
        outputStreamWriter.write('a'); ④
        outputStreamWriter.write('b');
        outputStreamWriter.write('c');
    }
}
```

```

        outputStreamWriter.close(); ⑤
    }
}

```

- ① Determinem el fitxer al que volem accedir.
- ② Accedim al fitxer seqüencialment i byte a byte amb un **FileOutputStream**, indicant la codificació de caràcters a utilitzar.
- ③ Creem un nou **OutputStreamWriter** a partir del **FileOutputStream** anterior.
- ④ Escrivim al fitxer caràcter a caràcter, en aquest cas passem una cadena que serà guardada caràcter a caràcter.
- ⑤ Tanquem el "writer".

30.2.1. Determinar la codificació de caràcters

La classe **OutputStreamWriter** permet triar la codificació de caràcters que s'utilitzarà per escriure al **OutputStream** encapsulat. Per exemple:

```

OutputStream outputStream = new FileOutputStream("c:\\\\data\\\noutput.txt");
OutputStreamWriter outputStreamWriter = new
OutputStreamWriter(outputStream, "ISO-8859-1"); // ISO Latin 1

```

30.2.2. Tancar un OutputStreamWriter

Cal alliberar els recursos i bloqueigs associats al fitxer en acabar d'utilitzar-lo, per fer-ho, cal cridar al mètode **close()**.

Tancar un **OutputStreamReader** tanca automàticament el **OutputStream** encapsulat.

30.3. Sobreescriure el fitxer o afegir dades al final

Per habilitar el mode d'addició n'hi ha prou en habilitar-lo pel **FileOutputStream** subjacent.

```

OutputStream outputStream = new FileOutputStream("c:\\\\data\\\\output.txt",
true); ①
OutputStreamWriter outputStreamWriter = new
OutputStreamWriter(outputStream, "utf-8");

```

- ❶ El paràmetre booleà a **true** habilita l'addició de dades al final del fitxer.

30.4. Classe java.io.InputStreamReader

C	InputStreamReader
●	InputStreamReader(in: InputStream)
●	InputStreamReader(in: InputStream, cs: Charset)
●	InputStreamReader(in: InputStream, dec: CharsetDecoder)
●	InputStreamReader(in: InputStream , charsetName: String)
●	close(): void
●	getEncoding(): String
●	read(): int
●	read(cbuf: char[], offset: int, length: int): int
●	ready(): boolean

Com a especialització de la classe **Reader** hi ha la classe **InputStreamReader**, la característica principal d'aquest "Reader" és que es construeix a partir d'una de les classes **Stream**, que recordem, servien per tractar els fluxos de dades byte a byte.

Per tant, l'objectiu d'aquesta classe és **adaptar un objecte capaç de llegir un flux de bytes a un objecte que es capaç de llegir el mateix flux de dades però caràcter a caràcter enlloc de byte a byte**.

Per exemple:

```
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;

public class FileTest {

    public static void main(String[] args) throws IOException {
        File f = new File("c:/data/input.txt"); ❶
        FileInputStream inputStream = new FileInputStream(f); ❷
        InputStreamReader inputStreamReader = new
        InputStreamReader(inputStream, "utf-8"); ❸
    }
}
```

```

int data = inputStreamReader.read();
while (data != -1) {
    char caracterLlegit = (char) data; ④
    data = inputStreamReader.read();
}

inputStreamReader.close();
}
}

```

- ①** Determinem el fitxer al que volem accedir
- ②** Accedim al fitxer seqüencialment i byte a byte amb un **FileInputStream**.
- ③** Creem un nou **InputStreamReader** a partir del **FileInputStream** anterior.
- ④** Ara podem accedir al fitxer caràcter a caràcter.



La classe **InputStreamReader** s'acostuma a utilitzar per a fitxers (o connexions de xarxa) on els bytes representen text. Per exemple, un fitxer de text amb els caràcters codificats en UTF-8. En aquest cas es pot utilitzar un **InputStreamReader** per encapsular un **FileInputStream** i així poder llegir el fitxer com a text.

30.4.1. Mètode read()

El mètode **read()** d'un **InputStreamReader** retorna un **int** que conté el valor del **char** llegit.

Fixeu-vos que **read()** retorna un **int** enllloc d'un **char** que és el què esperàvem. Això és així per permetre detectar el final de fitxer mentre es realitza una lectura seqüencial.

30.4.2. Final de fitxer

Recordeu que un **char** sempre és un **número positiu**.

Si el mètode **read()** retorna **-1**, cal un **int** per emmagatzemar el **-1**, vol dir que s'ha arribat al final del fitxer, en aquest cas es pot tancar el **InputStreamReader** cridant al seu mètode **close()**.

30.5. Selecció de la codificació de caràcters

A l'hora de crear un nou **InputStreamReader** es pot indicar quina serà la codificació de caràcters emprada, per exemple:

```
InputStream inputStream = new FileInputStream("c:/data/input.txt");
Reader inputStreamReader = new InputStreamReader(inputStream, "UTF-8");
```

Algunes de les possibles codificacions són: "US-ASCII", "ISO-8859-1", "UTF-8", "UTF-16"

30.5.1. Tancar un **InputStreamReader**

Cal alliberar els recursos i bloqueigs associats al fitxer en acabar d'utilitzar-lo, per fer-ho, cal cridar al mètode **close()**.

Tancar un **InputStreamReader** tanca automàticament el **InputStream** encapsulat.

30.6. Classe **java.io.FileReader** i **java.io.FileWriter**

La classe **FileReader** permet llegir el contingut d'un fitxer caràcter a caràcter, és una especialització de la classe **InputStreamReader** amb la única diferència que es construeix directament a partir d'un objecte **File** enlloc d'un objecte **InputStream**.

Per exemple, enlloc de fer:

```
File f = new File(".\\data\\input.txt");
FileInputStream fis = new FileInputStream(f);
InputStreamReader isr = new InputStreamReader(fis);
```

Fariem directament:

```
File f = new File(".\\data\\input.txt");
FileReader fr = new FileReader(f);
```

Pel demés el funcionament és exactament el mateix que el de la classe **InputStreamReader**.

I similarment, la classe **FileWriter** permet escriure caràcters a un fitxer. És una especialització de la classe **OutputStreamWriter** amb la única diferència que es construeix directament a partir d'un objecte **File** enlloc d'un objecte **OutputStream**.

Per exemple, enlloc de fer:

```
File f = new File(".\\data\\input.txt");
FileOutputStream fos = new FileOutputStream(f);
OutputStreamWriter osw = new OutputStreamWriter(fos);
```

Fariem directament:

```
File f = new File(".\\data\\input.txt");
FileWriter fr = new FileWriter(f);
```

Pel demés el funcionament és exactament el mateix que el de la classe **OutputStreamWriter**.

30.7. Sobreescrivir el fitxer o afegir dades al final

Quan es crea un **FileWriter** apuntant a un fitxer que ja conté dades cal decidir si es vol sobreescrivir el fitxer o bé afegir les noves dades al final.

Per fer-ho cal passar un paràmetre addicional a l'hora de crear l'objecte **FileWriter**, aquest paràmetre és un **boolean** que indica si el fitxer s'obre per a sobreescritura, **false**, o bé per afegir dades, **true**.

Per exemple:

```
FileWriter fileWriter = new FileWriter("c:/data/output.txt", true); //  
afegeix dades al fitxer  
  
FileWriter fileWriter = new FileWriter("c:/data/output.txt", false); //  
sobreescriva el fitxer
```



Si no s'especifica el paràmetre booleà el fitxer s'obre sobreescrivint les dades originals.

```
FileWriter fileWriter = new FileWriter("c:/data/  
output.txt");
```

31

Lectura i escriptura de fitxers de text a alt nivell

Treballar amb fitxers a baix nivell pot ser útil a vegades però normalment voldrem classes per permetir llegir i escriure tipus de dades primitius directament, enlloc de només un caràcter cada vegada.

Per fer-ho, utilitzarem la classe **Scanner** per la lectura de fitxers de text i la classe **PrintWriter** per la seva escriptura.

31.1. Classe Scanner

Recordem que l'objecte Scanner encapsula un flux de dades d'entrada proporcionat just a la creació de l'objecte. Si es vol llegir dades des de la consola caldrà encapsular el flux **System.in**.

```
Scanner in = new Scanner(System.in);
```

De forma similar es pot llegir el contingut d'un fitxer de text amb un objecte Scanner, només fa falta obtenir un flux de dades de lectura al fitxer, això ho proporciona la classe **File**. Per tant, caldrà:

```
File inputFile = new File("input.txt");
Scanner in = new Scanner(inputFile);
```

Per detectar el final de fitxer es pot fer amb l'ajuda dels mètodes **hasNext()** i un bucle **while**.

Per exemple, per llegir dades d'un fitxer de text amb números es podria fer:

```
File inputFile = new File("input.txt");
Scanner in = new Scanner(inputFile);

while (in.hasNextDouble()) {
    double valor = in.nextDouble();
    // Processar el valor
}
```

Finalment caldria alliberar els recursos dedicats a mantenir el fitxer de text obert, per fer-ho és important cridar el mètode **close()** de l'objecte Scanner.

```
in.close();
```

Anem a veure un exemple sencer d'utilització de la classe Scanner per a la lectura d'un fitxer de text.

31.2. Exemple lectura d'un fitxer de text mitjançant la classe Scanner

Volem llegir fitxers amb la següent estructura:

```
Pere López Antuno 8 María Ferrer Ortuño 5 Eva Gusi Pons
```



Hi ha un problema addicional amb el que hem de tractar. Si el fitxer al que volem accedir no existeix es produirà una excepció de tipus **IOException** quan es construeixi l'objecte Scanner.

El problema és que el compilador insisteix en que especificuem què és el que ha de fer el programa quan es produïx aquesta excepció, si no li especificuem el programa no es deixa compilar.

Per sortir del pas indicarem al compilador que si no es troba el fitxer indicat es dispari l'excepció en forma d'error d'execució, s'interrompi l'execució i es mostri la informació de l'excepció per la sortida estàndard.

Exemple lectura d'un fitxer de text mitjançant la classe Scanner

Per fer-ho cal declarar la funció **main** de la següent manera:

```
public static void main(String[] args) throws  
    IOException {  
}
```

Cal notar que aquesta no és una manera bona de procedir, el que voldrem aconseguir serà gestionar l'excepció i gestionar la condició de l'error, no volem la interrupció definitiva del codi.

```
import java.io.File;  
import java.io.IOException;  
import java.util.Scanner;  
  
public class ReadData {  
  
    public static void main(String[] args) throws IOException {  
  
        File file = new File("notes.txt");  
        Scanner in = new Scanner(file);  
  
        while (in.hasNext()) {  
            String nom = in.next();  
            String primerCognom = in.next();  
            String segonCognom = in.next();  
            int nota = in.nextInt();  
            System.out.println(nom + " " + primerCognom + " " +  
segonCognom + " " + nota);  
        }  
  
        // No oblidem tancar el fitxer!!  
        in.close();  
    }  
}
```



Què passaria si el nom d'un alumne fos Josep Maria?

31.3. Mètodes de la classe java.util.Scanner

C	Scanner
●	Scanner(source: File)
●	Scanner(source: String)
●	close()
●	hasNext(): boolean
●	next(): String
●	nextLine(): String
●	nextByte(): byte
●	nextShort(): short
●	nextInt(): int
●	nextLong(): long
●	nextFloat(): float
●	nextDouble(): double
●	useDelimiter(pattern: String):Scanner

Scanner(source: File)

Crea un objecte Scanner a partir del fitxer especificat.

Scanner(source: String)

Crea un objecte Scanner a partir de la cadena especificada.

close()

Tanca l'objecte Scanner.

hasNext(): boolean

Retorna true si el Scanner té més dades per ser llegides.

next(): String

Retorna el següent valor com a String.

nextLine(): String

Retorna un String fins a un separador de línia.

nextByte(): byte

Retorna el següent valor com a byte.

nextShort(): short

Retorna el següent valor com a short.

nextInt(): int

Retorna el següent valor com a int.

nextLong(): long

Retorna el següent valor com a long.

nextFloat(): float

Retorna el següent valor com a float.

nextDouble(): double

Retorna el següent valor com a double.

useDelimiter(pattern: String):Scanner

Estableix quin és el patró separador de valors i retorna el propi Scanner.

31.4. Classe java.io.PrintWriter

La classe **PrintWriter** permet escriure dades **amb format** al **Writer** subjacent. Per exemple, permet escriure dades **int**, **long** i d'altres tipus primitius com a text enlloc d'escriure directament el seu valor en bytes.

És una classe molt útil per generar informes o documents similars on hi ha la necessitat de barrejar text i números. La classe **PrintWriter** té els mateixos mètodes que la classe **PrintStream** (System.out és un PrintStream) amb excepció dels que permeten escriure bytes directament.

31.5. Utilització d'un objecte PrintWriter

En primer lloc crearem un objecte **PrintWriter** a partir d'un objecte **File**.

```
File outputFile = new File("output.txt");
PrintWriter out = new PrintWriter(outputFile);
```

A continuació podem afegir dades al fitxer utilitzant els mètodes print proporcionats per PrintWriter.

```
out.println("Això és una prova");
out.printf("Total: %.2f\n", total);
```

I finalment caldrà tancar el flux d'escriptura per alliberar els recursos associats i per garantir que tota la informació passada a l'objecte es traspassa efectivament al fitxer..

```
out.close();
```

Exemple d'utilització:

```
File file = new File("d:\\data\\report.txt");
FileWriter writer = new FileWriter(file);
PrintWriter printWriter = new PrintWriter(writer);

printWriter.print(true);
printWriter.print((int) 123);
printWriter.print((float) 123.456);

printWriter.printf(Locale.UK, "Text + data: %1$", 123);

printWriter.close();
```

31.6. Mètodes de la classe java.io.PrintWriter

java.io

C PrintWriter

- PrintWriter(file: File)
- PrintWriter(file: File, charset: String)
- PrintWriter(out: OutputStream)
- PrintWriter(out: OutputStream, autoFlush: boolean)
- PrintWriter(fileName: String)
- PrintWriter(fileName: String, charset: String)
- PrintWriter(out: Writer)
- PrintWriter(out: Writer, autoFlush: boolean)

- PrintWriter(filename: String)
- print(s: String): void
- PrintWriter(file: File)
- print(c: char): void
- print(cArray: char[]): void
- print(i: int): void
- print(l: long): void
- print(f: float): void
- print(d: double): void
- print(b: boolean): void

També conté les versions sobrecarregades dels mètodes println.
També conté les versions sobrecarregades dels mètodes printf.

PrintWriter(filename: String)

Crea un objecte PrintWriter pel nom de fitxer especificat.

print(s: String): void

Escriu una cadena al fitxer.

PrintWriter(file: File)

Crea un objecte PrintWriter pel objecte file especificat.

print(c: char): void

Escriu un char al fitxer.

print(cArray: char[]): void

Escriu una matriu de char al fitxer.

print(i: int): void

Escriu un int al fitxer.

print(l: long): void

Escriu un long al fitxer.

print(f: float): void

Escriu un float al fitxer.

print(d: double): void

Escriu un double al fitxer.

print(b: boolean): void

Escriu un boolean al fitxer.

També conté les versions sobrecarregades dels mètodes println. També conté les versions sobrecarregades dels mètodes printf

El mètode println i printf funciona igual que el mètode de mateix nom de System.out.

31.7. Exemple d'utilització de PrintWriter

El següent exemple obra un fitxer de text i en fa una còpia modificant totes les lletres a majúscula.

```
import java.io.File;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Scanner;

public class PrintWriterTest {

    public static void main(String[] args) throws IOException {

        Scanner console = new Scanner(System.in);
        System.out.print("Input file: ");
        String inputFileName = console.next();
        System.out.print("Output file: ");
        String outputFileName = console.next();

        File inputFile = new File(inputFileName);
        Scanner in = new Scanner(inputFile);
        PrintWriter out = new PrintWriter(outputFileName);

        while (in.hasNextLine()) {
```

```
        String linia = in.nextLine();
        out.println(linia.toUpperCase());
    }

    in.close();
    out.close();
}
}
```

31.8. Llegir un fitxer de text per paraules

L'objectiu d'aquest apartat és fer un programa que donada una frase retorna les paraules.

Considerem el bucle següent:

```
while (in.hasNext()) {
    String input = in.next();
    System.out.println(input);
}
```

Si l'usuari introduceix:

```
Pablito clavó un clavito, ¿qué clavito clavó Pablito?, el clavito que
Pablito clavó, era el clavito de Pablito.
```

la sortida serà:

```
Pablito
clavó
un
clavito,
¿qué
clavito
clavó
Pablito?,
el
clavito
que
Pablito
clavó,
```

```
era  
el  
clavito  
de  
Pablito.
```

El problema és que apareixen els signes de puntuació com si formessin part de les paraules que es volen separar.

A vegades es volen llegir les paraules i descartar qualsevol cosa que no sigui una lletra, per fer-ho es pot utilitzar el mètode **useDelimiter** de la classe Scanner i una **expressió regular**.

Per exemple:

```
in.useDelimiter("[^A-Za-z]+");  
while (in.hasNext()) {  
    String input = in.next();  
    System.out.println(input);  
}
```

Expressions regulars

Les expressions regulars descriuen patrons de lletres.

Per exemple, els números enters positius tenen un patró simple, poden contenir un o més dígits. L'expressió regular que descriu els números és [0-9]+.

El conjunt [0-9] denota qualsevol dígit entre 0 i 9 i el símbol + vol dir "**un o més**".

Les comandes de cerca dels editors de text solen entendre les expressions regulars, de fet, moltes altres utilitats fan ús d'aquestes expressions per localitzar patrons de text.

Per exemple el programa **grep** que significa "global regular expression print" n'és un exemple clar.

Per exemple,

```
grep -E [0-9]+ Test.java
```

Mostraria totes les línies del fitxer Test.java que continguin seqüències de números. No massa útil... Les línies amb noms de variables amb números, com ara x1, també es mostrarien.

Eliminem la possibilitat de mostrar números precedits immediatament per una lletra:

```
grep -E [^A-Za-z][0-9]+ Test.java
```

El conjunt [^A-Za-z] indica qualsevol lletra que **no** està dins dels rangs A-Z i a-z.

Per exemple, la següent expressió regular podria identificar correus electrònics:

```
\b[A-Za-z0-9]+@[A-Za-z0-9]+\.[A-Za-z]{2,6}\b
```

Les expressions regulars són complexes i dependents del llenguatge i queden fora d'aquest curs.

31.9. Classe Scanner i la codificació de caràcters

Al crear un objecte Scanner es pot indicar la codificació de caràcters utilitzada per interpretar la lectura de les dades.

Per exemple:

```
Scanner in = new Scanner(System.in, "Windows-1252");
```

Només cal indicar el nom de la codificació com a segon paràmetre a la creació de l'objecte.

31.10. Llegir un fitxer de text caràcter a caràcter

A vegades voldrem llegir un fitxer caràcter a caràcter, en aquets cas tenim dues opcions.

1. Prenem tota la línia amb el mètode **nextLine()** de la classe Scanner i manipulem la línia amb els mètodes de les classes **String** i **Character**.
2. Establim el mètode **useDelimiter** de la classe Scanner amb una cadena buida.

```
Scanner in = new Scanner(.....);
in.useDelimiter("");
```

31.11. Classificar caràcters

Pot ser interessant recordar alguns dels mètodes de la classe **Character** que permeten distingir un tipus de caràcter d'un altre.

isDigit, **isLetter**, **isUpperCase**, **isLowerCase**, **isWhiteSpace**, ...

31.12. Llegir un fitxer de text línia a línia

Els mètodes **substring** i **trim** de la classe String poden ser útils per fer un tractament de les dades d'un fitxer de text línia a línia.

31.13. Escanejar una cadena

Es pot utilitzar la classe **Scanner** en una variable cadena.

```
linia = "Pere Lòpez Antuno 8 Maria Ferrer Ortúñ 5 Eva Gusi Pons";
Scanner scannerLinia = new Scanner(linia);
String nomComplert = "";
while (!scannerLinia.hasNextInt()) {
    nomComplert = nomComplert + " " + scannerLinia.next();
}
```

Bibliografia

Llibres

Allen B. Downey. 'Think Java'. O'Reilly Media. 2016. ISBN 978-1-491-92956-8

Kishori Sharan. 'Beginning Java 8 Fundamentals'. Apress. 2014. ISBN 978-1-430-26653-2.

Kishori Sharan. 'Beginning Java 8 Language Features: Lambda Expressions, Inner Classes, Threads, I/O, Collections, and Streams'. Apres. 2014. ISBN 978-1-430-26659-4

Robert C. Martin. 'Código limpio: Manual de estilo para el desarrollo ágil de software'. Anaya Multimedia. 2012. ISBN 978-8-441-53210-6

Daniel Liang. 'Introduction to Java Programming, Comprehensive Version'. 9th Edition. Pearson. 2012. ISBN 978-0-132-93652-1

Robert Liguori. 'Java 8 Pocket Guide'. O'Reilly Media. 2014. ISBN 978-1-491-90086-4

Allan Vermeulen. 'The Elements of Java Style'. 9th Edition. Cambridge University Press. 2007. ISBN 978-0-521-77768-1.

Stuart Reges. 'Building Java Programs: A Back to Basics Approach' 2nd Edition. Pearson. 2010. ISBN 978-0-136-09181-3

Walter J. Savitch. 'Java: An Introduction to Problem Solving & Programming'. 6th Edition. Pearson. 2011. ISBN 978-0-132-16270-8

Robert Sedgewick. 'Algorithms'. 4th Edition. Addison-Wesley Professional. 2011. ISBN 978-0-321-57351-3

Joshua Bloch. 'Effective Java'. 2nd Edition. Addison-Wesley. 2008. ISBN 978-0-321-35668-0

Part IV. UF4

Programació orientada a objectes. Fonaments

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.

— C.A.R. Hoare *Informàtic Britànic inventor de la referència null*

Sumari

32. Resultats d'aprenentatge i criteris d'avaluació	309
33. Continguts	311
34. Classes i Objectes	313
34.1. Què és una classe i què és un objecte?	314
34.2. Declarar una classe	315
34.3. Les classes defineixen nous tipus de dades per referència	315
34.4. Crear instàncies d'una classe	316
34.5. Declarar camps en una classe	316
34.6. El tipus null	318
34.7. L'operador . per accedir als camps d'una classe	319
34.8. Inicialització per defecte dels camps	320
34.9. Revisió de la gestió de la memòria en Java	320
34.9.1. Quina de les estructures utilitzar en Java?	322
34.10. Tots les classes són tipus per referència	322
34.11. L'operador d'assignació i els tipus per referència	322
34.11.1. Assignació i tipus primitius	322
34.11.2. Assignació i tipus per referència	323
34.12. Modificadors d'accés per una classe	324
34.13. Revisió de la clàusula import	326
34.14. Declaració de mètodes en una classe	327
34.15. Variables locals	327
34.16. Mètodes d'una instància i mètodes d'una classe	328
34.17. Invocar un mètode	329
34.18. Constants	330
34.19. Revisió del mètode main	331
34.20. Paraula clau this	331
34.21. Modificadors d'accés pels membres d'una classe	333
34.22. Representació de classes en UML	333
34.23. Primer principi de la programació orientada a objectes, l'encapsulació i l'amagament de la informació	335
34.24. L'encapsulació a la pràctica	337
34.25. Constructors	338
34.26. Sobrecàrrega de constructors	340
34.27. Cridar un constructor des d'un altre constructor	341

34.28. Constructor per defecte	343
34.29. Representació de les relacions entre classes en UML	344
34.30. Representació de les relacions entre classes en UML - Exemple	347
34.31. Exemples	347
34.31.1. CercleSimple	347
34.31.2. TV	349
35. Principis de disseny de classes	353
35.1. Príncipi de Cohesió	353
35.2. Príncipi de Consistència	353
35.3. Príncipi d'encapsulació	354
35.4. Príncipi de Claredat	354
36. La classe Object (Primera part)	355
36.1. Totes les instàncies són un objecte	355
36.2. Els mètodes de la classe Object	355
36.3. Quina és la classe d'un objecte?	357
36.4. Comparar objectes per saber si són iguals	358
36.4.1. Especificacions alhora de reimplementar el mètode equals()	360
36.5. Calcular el "hash code" d'un objecte	361
36.5.1. Per què els objectes mantenen un "hash" code?	361
36.5.2. Quan cal reimplementar el mètode hashCode()	362
36.5.3. Exemple	362
36.6. Representació d'un objecte amb una cadena	364
36.7. Exemple - Una classe completa	365
36.8. Finalitzar un Objecte - Deprecated JDK11	367
37. Herència	369
37.1. Superclasses i subclasses	369
37.2. Representació de la relació d'herència en UML	370
37.3. Paraula clau extends	371
37.4. La classe Object és la superclasse per defecte	371
37.5. Herència i la relació jeràrquica	372
37.6. Quines coses són heretades per una subclasse?	373
37.7. Herència múltiple	374
37.8. Upcasting i Downcasting	375
37.8.1. Upcasting	375

37.8.2. Downcasting	375
37.9. Operador instanceof	376
37.10. Compte amb l'operador instanceof i el mètode equals	377
37.11. Sobreescritura de mètodes	379
37.12. Anotació @override	382
37.13. Cridar un mètode sobreescrit des d'una classe filla, paraula clau super	382
37.14. Upcasting i sobreescritura, "overriding"	383
37.15. Les variables en Java no es poden sobreescrivir	384
37.16. Accés a mètodes sobreescrits	386
37.17. Sobrecàrrega de mètodes, "overloading"	387
37.18. L'erència i els constructors	387
37.19. Amagar mètodes	393
37.20. Amagar camps	393
37.21. Deshabilitar l'erència	393
37.22. Paraula clau final	394
37.22.1. Variables final	394
37.22.2. Mètodes final	394
37.22.3. Classes final	394
37.23. Objectes immutables	396
37.23.1. Quins avantatges tenen els objectes immutables	397
37.23.2. Estratègies per a definir objectes immutables	397
37.24. Classes i mètodes abstractes	402
37.25. Composició per sobre de l'erència	404
38. Interfícies	407
38.1. Tenim un problema...	407
38.2. Una solució mitjançant interfícies	412
38.3. Declarar una interfície	414
38.4. Mètodes estàtics a les interfícies	415
38.5. Mètodes per defecte a les interfícies	415
38.6. Declaració d'interfícies niuades	415
38.7. Una interfície defineix un nou tipus de dades	415
38.8. Implementar una interfície	415
38.9. Interfícies en UML	416
38.10. Implementar múltiples interfícies	417
38.11. Implementació parcial d'interfícies	418

38.12. Les interfícies admeten herència (i herència múltiple)	418
38.13. El Polimorfisme revisat	419
39. Principis de disseny a la programació orientada a objectes	421
39.1. Hem de sospitar que el nostre codi té problemes si...	421
39.2. Príncipi de responsabilitat única	421
39.3. Príncipi d'obert-tancat	422
39.4. Príncipi de substitució de Liskov	424
39.5. Príncipi de la segregació d'interfícies	425
39.6. Príncipi de la inversió de les dependències	426

32

Resultats d'aprenentatge i criteris d'avaluació

1. Escriu i prova programes senzills, reconeixent i aplicant els fonaments de la programació orientada a objectes.
 1. Defineix objectes a partir de classes predefinides.
 2. Utilitza mètodes i propietats dels objectes.
 3. Escriu crides a mètodes estàtics.
 4. Utilitza paràmetres a la crida a mètodes.
 5. Incorpora i utilitza llibreries d'objectes.
 6. Utilitza constructors.
 7. Distingeix dades estàtiques de dades dinàmiques.
 8. Reconeix els mecanismes de destrucció i/o finalització d'objectes.
 9. Reconeix els mecanismes d'alliberament de memòria.
10. Utilitza l'entorn integrat de desenvolupament en la creació i compilació de programes simples.
2. Desenvolupa programes organitzats en classes analitzant i aplicant els principis de la programació orientada a objectes.
 1. Reconeix la sintaxi, l'estructura i els components típics d'una classe.
 2. Defineix classes.
 3. Defineix propietats i mètodes.
 4. Crea constructors.

-
- 5. Crea destructors i/o mètodes de finalització.
 - 6. Desenvolupa programes que instancien i utilitzen objectes de les classes creades anteriorment.
 - 7. Utilitza mecanismes per controlar la visibilitat de les classes i dels seus membres.
 - 8. Defineix i utilitza classes heretades.
 - 9. Crea i utilitza mètodes estàtics.
 - 10. Crea i utilitza conjunts i llibreries de classes.
3. Desenvolupa programes aplicant característiques avançades dels llenguatges orientats a objectes i de l'entorn de programació.
- 1. Identifica els conceptes d'herència, superclasse i classe subclasse.
 - 2. Utilitza modificadors per bloquejar i forçar l'herència de classes i mètodes.
 - 3. Reconeix la incidència dels constructors en l'herència.
 - 4. Reconeix la incidència dels destructors i/o mètodes de finalització en l'herència.
 - 5. Crea classes heretades que sobreescriguin la implementació de mètodes de la superclasse.
 - 6. Coneix l'existència de l'herència múltiple i els problemes derivats.
 - 7. Dissenya i aplica jerarquies de classes.
 - 8. Prova i depura les jerarquies de classes.
 - 9. Realitza programes que implementin i utilitzin jerarquies de classes.
 - 10. Comenta i documenta el codi.
 - 11. Entén, defineix i implementa interfícies.

33

Continguts

1. Introducció a la programació orientada a objectes:

1. Tipus primitius de dades.
2. Característiques dels objectes.
3. Definició d'objectes.
4. Taules de tipus primitius davant de taules d'objectes.
5. Utilització de mètodes.
6. Utilització de propietats.
7. Utilització de mètodes estàtics.
8. Constructors.
9. Memòria: gestió dinàmica davant de gestió estàtica; possibilitats del llenguatge.

10 Destrucció i/o finalització d'objectes i alliberament de memòria.

2. Desenvolupament de programes organitzats en classes:

1. Concepte de classe. Estructura i membres.
2. Creació d'atributs.
3. Creació de mètodes.
4. Sobrecàrrega de mètodes.
5. Creació de constructors.
6. Creació de destructors i/o mètodes de finalització.
7. Ús de classes i objectes. Visibilitat.

-
8. Conjunts i llibreries de classes.
 3. Utilització avançada de classes en el disseny d'aplicacions:
 1. Composició de classes.
 2. Herència.
 3. Jerarquia de classes: superclasses i subclasses.
 4. Classes i mètodes abstractes i finals.
 5. Sobreescriptura de mètodes.
 6. Herència i constructors/destructors/mètodes de finalització.
 7. Interfícies.

34

Classes i Objectes

Els paradigmes de programació més utilitzats actualment són la **programació procedural** i la **programació orientada a objectes**.

El primers llenguatges de programació eren procedurals, és a dir:

- Un programa estava format per un **conjunt de procediments**.
- Un procediment és un conjunt d'instruccions que, conjuntament, realitzen una tasca específica, ja sigui recopilar informació de l'usuari, manipular informació emmagatzemada a la memòria de l'ordinador o bé realitzar determinats càlculs per obtenir la solució a un problema.

Els procediments operen sobre **dades** que estan separades dels propis procediments. **En un programa procedural les dades a tractar es solen passar d'un procediment a l'altre.**

Aquesta separació entre dades i codi sovint genera problemes:

- Les dades estan emmagatzemades en diferents formats consistents en variables i estructures més complexes creades a partir de variables.
- Els procediments que manipulen aquestes dades han d'haver estat dissenyats tenint en compte el format específic de les dades que han de tractar.

Però que passa si **el format de les dades s'altera d'alguna manera**, un requeriment que ha canviat, una nova funcionalitat, etc..

En aquest cas caldrà modificar **tots** els procediments que interaccionen d'alguna manera amb les dades el format de les quals ha estat modificat, cosa que propicia la introducció de nous bugs al programa.

Aquests problemes van influenciar el canvi de paradigma des de la programació procedural a la programació orientada a objectes. De la mateixa manera que la programació procedural està centrada en crear **procediments**, la programació orientada a objectes està centrada en la creació d'**objectes**.

Un **objecte** és una entitat de software que conté **dades i procediments**.

- Les dades contingudes dins de l'objecte reben el nom d'**atributs** de l'objecte
- Els procediments reben el nom de **mètodes** de l'objecte.

Un objecte és, conceptualment, una unitat auto-continguda consistent en **dades i procediments**.

La programació orientada a objectes soluciona el problema de la separació de codi/dades mitjançant la **encapsulació i l'amagament de les dades**.

L'**encapsulació** fa referència al fet de combinar procediments i dades dintre d'una mateixa estructura.

L'**amagament de les dades** fa referència a l'habilitat de l'objecte d'amagar les **dades** des de l'exterior de l'objecte.

34.1. Què és una classe i què és un objecte?

Un objecte representa una entitat del món real que es pot identificar de manera unívoca. Per exemple, una taula, un alumne, una casa, un préstec.

Un objecte té una **identitat única**, un **estat** i un **comportament**.

- L'**estat** d'un objecte, representat amb el nom de **propietats** o d'**atributs**, està representat per camps de dades amb els seus valors actuals. Per exemple, un **Cercle** tindrà un estat **radi**, i un **Rectangle** un estat **alçada i amplada**.
- El **comportament** d'un objecte, definit amb el nom de les accions de l'objecte, està definit per **mètodes**. Invocar un mètode d'un objecte és demanar a l'objecte que realitzi una acció. Per exemple, un Cercle tindrà un comportament **obtenirArea()** o **obtenirPerimetre()**.

Els objectes del mateix tipus es defineixen utilitzant la mateixa classe. Una classe és un patró que defineix quins camps de dades i quins mètodes tindrà l'objecte en qüestió. En cap cas defineix quins valors concrets tindrà el seu estat.

Un objecte és una instància d'una classe. Es poden crear varies instàncies de la mateixa classe. Crear una instància d'una classe rep el nom d'**instanciar**. Els termes **objecte** i **instància** són intercanviables.

Una classe de Java utilitza **variables** per a definir l'estat i **funcions** per a definir el comportament.

Una classe proporciona uns mètodes especials, anomenats **constructors**, que es criden en el moment en que un objecte es crea, aquests mètodes estan dissenyats per realitzar accions d'inicialització d'un objecte com per exemple inicialitzar l'estat de l'objecte.

34.2. Declarar una classe

La sintaxi general per a declarar una classe és:

Sintàxis general de la declaració d'una classe.

```
modificadors class nom-classe {
    // El cos de la classe
}
```

modificadors

paraules clau que associen significats especials a la declaració de classe. Una declaració de classe pot tenir cap o més modificadors.



Per conveni els **noms de les classes** en Java comencen en **majúscula** i es segueix amb notació **camelCase**.

34.3. Les classes defineixen nous tipus de dades per referència

Una classe **defineix un nou tipus de dades** i per tant es poden declarar variables d'aquest nou tipus, per exemple:

```
public class Persona {

}

public class Main {
```

```
public static void main(String[] args) {
    Persona p1;
    Persona p2;
}
```

Aquests nous tipus de dades són indistingibles dels tipus de dades per referència definits a les llibreries de classes Java, com per exemple, **String**, **Scanner** o **PrintWriter** i per tant es poden utilitzar com a **paràmetres d'entrada** en un mètodes, com a **valor de retorn** d'un mètode o per **crear-ne matrius**.

Per exemple,

```
public static Persona obtenerGuanyador(Persona[] p) {
    // Fer un sorteig i retornar la Persona guanyadora
}
```

34.4. Crear instàncies d'una classe

La sintaxi general per crear una instància d'una classe és:

Creació d'una instància amb l'operador new.

```
new crida-al-constructor-de-la-classe
```

Per exemple:

```
Punt2D p1 = new Punt2D();
```

En l'exemple anterior **Punt2D()** és una crida al constructor de la classe **Punt2D**.

Quan una nova instància d'una classe és creada, l'operador **new** reserva memòria per a cada camp de la nova instància.

Com que no s'ha afegit cap constructor a la classe el compilador de Java n'afegeix un automàticament. Aquest constructor s'anomena **constructor per defecte**.

34.5. Declarar camps en una classe

Els camps d'una classe representen propietats (a vegades s'anomenen atributs) dels objectes de la classe.

Els camps es declaren dintre del cos de la classe, la sintaxi general és:

Sintàxis general de la declaració dels camps d'una classe.

```
modificadors class nom-classe {  
    // Declaració d'un camp  
    modificadors tipus-de-dada nom-camp = valor-inicial  
}
```



Per conveni els **noms dels camps** en Java comencen en **minúscula** i es segueix amb notació **camelCase**.

Per exemple:

```
class Punt2D {  
    float coordenadax;  
    float coordenadaY;  
}
```

En Java es poden declarar dos tipus de camps per una classe:

Camps que pertanyen a la Classe

- Totes les instàncies de la classe comparteixen la mateixa variable.
- Es pot modificar el valor de la variable des d'una instància o directament des de la pròpia classe.
- Es declaren amb la paraula clau **static**.
- També reben el nom de variables estàtiques.

Camps que pertanyen a la instància

- Cadascuna de les instàncies de la classe té la seva versió de la variable.
- Modificar el camp en una objecte no fa que canviï el valor en un altre de la mateixa classe.

Exemples:

```
class Punt2D {  
    float coordenadax;  
    float coordenadaY;
```

```

    static int contador;
}

Punt2D p1 = new Punt2D();
p1.coordenadaX = 11;
p1.coordenadaY = 21;
p1.contador = 1; // És equivalent a Punt2D.contador = 1;
Punt2D p2 = new Punt2D();
p2.coordenadaX = 12;
p2.coordenadaY = 22;
p2.contador = 2; // És equivalent a Punt2D.contador = 2;

System.out.println("(" + p1.coordenadaX + ", " + p1.coordenadaY);
System.out.println("(" + p2.coordenadaX + ", " + p2.coordenadaY);
System.out.println(p1.contador);
System.out.println(p2.contador);
System.out.println(Punt2D.contador);

```

```

(11, 21)
(12, 22)
2
2
2

```

34.6. El tipus null

Cada classe de Java defineix un tipus per referència. Java té un tipus per referència especial anomenat **null**. El tipus null no té nom i per tant no es pot crear una instància d'aquesta classe.

El tipus per referència **null** només conté un valor definit per Java que és el literal **null**.

El tipus per referència **null** és compatible amb qualsevol altre tipus per referència, és a dir, es pot assignar el valor **null** a qualsevol variable per referència. A la pràctica el el valor **null** assignat a una variable significa que la variable no fa referència a cap objecte.

Per exemple:

```
Punt2D p1 = null;
```

Es pot utilitzar el literal *null* amb els operadors de comparació per comprovar igualtat i no igualtat.

```
if (p1 == null) {  
    // p1 fa referència a null i no es pot utilitzar  
}  
  
if (p1 != null) {  
    // p1 té una referència a un objecte  
}
```



El literal **null** només és compatible amb tipus per referència, utilitzar **null** en un tipus primitiu donarà un error de compilació.

34.7. L'operador . per accedir als camps d'una classe

L'operador **.** s'utilitza per fer referència a les variables d'una instància. La seva sintaxi general és:

Crida de camps d'una classe a través d'una instància.

```
variable-per-referència.nom-variable-de-la-instància
```

Si el camp és *static* es pot accedir a la variable directament a través de la classe.

Crida de camps d'una classe a través de la classe.

```
nom-de-la-classe.nom-variable-estàtica
```

Aquesta notació s'utilitza tant per accedir a un camp com per a llegir-ne el valor.

Exemples:

Exemple 1.

```
Punt2D p1 = new Punt2D();  
p1.coordenadaX = 11;  
p1.coordenadaY = 21;
```

Exemple 2.

```
Punt2D.contador = 1
```

Exemple 3.

```
System.out.println(p1.coordenadaX);
System.out.println(p1.coordenadaY);
System.out.println(p1.contador);
System.out.println(p2.contador);
System.out.println(Punt2D.contador);
```

34.8. Inicialització per defecte dels camps

Tots els camps d'una classe, estàtics o no, **s'inicialitzen automàticament al seu valor per defecte**. Aquest valor depèn del seu tipus de dada.

- Un camp **numèric** (**byte, short, char, int, long, float, i double**) s'inicialitza a **0**.
- Un camp **boolean** s'inicialitza a **false**.
- Una **referència** s'inicialitza a **null**

34.9. Revisió de la gestió de la memòria en Java

Recordeu d'ufs anteriors que quan s'executa una aplicació es reserva una secció consecutiva de memòria amb la següent estructura:



Codi

És on es carregà el codi en execució.

Global

S'utilitza per emmagatzemar variables i apuntadors globals o estàtics.

Stack

La **pila** és una regió especial de la memòria de l'ordinador que emmagatzema variables temporals creades per a cada funció, en particular per la funció **main**.

Té una estructura de tipus **LIFO**, "Last Input First Output", que està optimitzada per la CPU.

Quan una funció declara una nova variable aquesta és afegida "al damunt" de l'estructura LIFO. Cada cop que una funció finalitza **totes** les variables declarades en aquella funció s'eliminen i s'allibera l'espai de memòria que ocupaven. L'àrea de memòria alliberada queda disponible per a noves variables declarades a noves funcions.

L'avantatge principal d'utilitzar la pila per emmagatzemar variables és que la gestió de la memòria és automàtica i el fet de treballar en una estructura de pila fa que la creació i alliberament de la memòria sigui molt ràpida.

Per altra banda:

- La quantitat de memòria dedicada a la pila està limitada.
- El tractament LIFO que fa la pila impedeix que un element situat al mig de la pila pugui créixer, solaparia la memòria del següent element, per tant, aquesta estructura només pot emmagatzemar elements que no canvien de mida durant la seva existència.

Heap

El "**heap**" és una regió de memòria no gestionada automàticament, que a diferència del "**stack**" no té restriccions en la mida de les variables que pot contenir, llevat de la pròpia mida de la RAM de l'equip.

L'estructura del "heap" és "lliure", de manera que es pot emmagatzemar un valor a qualsevol direcció de memòria de l'estructura, això fa que:

- Abans d'emmagatzemar un valor al "heap" caldrà buscar i trobar un espai de memòria de suficient capacitat. Cosa que no feia falta al "stack" i per tant repercutix negativament al rendiment.
- Els elements emmagatzemats al "heap" no s'eliminen automàticament al sortir de la funció que els utilitza, en alguns llenguatges s'han d'alliberar manualment, C, C++, i en d'altres un element específic de la màquina virtual, anomenat **recolecció de brossa** serà l'encarregat de fer-ho quan detecti que no s'utilitzen més.

- Com que la reserva i l'alliberament de la memòria és manual, la memòria tendeix a quedar fragmentada, cosa que té un impacte negatiu al rendiment.

34.9.1. Quina de les estructures utilitzar en Java?

Java no permet al programador triar directament quina de les estructures de memòria utilitzar.

El **tipus de dades** que es vol emmagatzemar a la memòria determina quina de les estructures s'utilitza.

- Els **tipus de dades primitius** s'emmagatzemen al "stack".
- Els **tipus de dades per referència** s'emmagatzemen al "heap" i es manté la direcció de memòria on s'ubiquen al "stack".

34.10. Tots les classes són tipus per referència

Sempre que treballem amb objectes, aquests s'emmagatzemen al **heap**.

34.11. L'operador d'assignació i els tipus per referència

L'operador d'assignació, `=`, té un comportament diferent en funció de si els operadors són tipus per valor o tipus per referència. Vegem-ho amb un exemple:

34.11.1. Assignació i tipus primitius

Considerem el següent codi i imaginem la gestió de memòria entre en el stack i en el heap.

En primer lloc vegem un exemple amb tipus primitius:

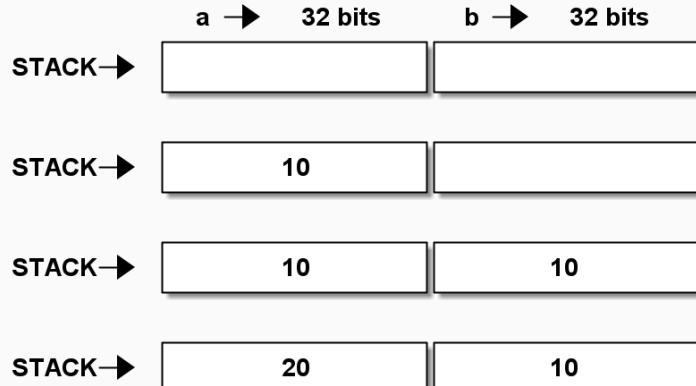
```
public class ExempleStack {  
  
    public static void main(String[] args) {  
        int a;  
        int b;  
  
        a = 10;  
  
        b = a;
```

```
a = 20;

System.out.printf("a: %d, b: %d%n", a, b);
}
```

a: 20, b: 10

Seguint el flux d'execució del programa obtenim:



34.11.2. Assignació i tipus per referència

I ara un exemple similar però amb tipus per referència:

```
class ValorInt {

    int valor;
}

public class ExempleStack {

    public static void main(String[] args) {
        ValorInt a;
        ValorInt b;
```

```

    a = new ValorInt();
    b = new ValorInt();
    a.valor = 10;

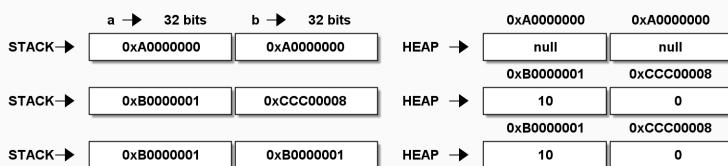
    b = a;

    System.out.printf("a: %d, b: %d%n", a.valor, b.valor);
}
}

```

a: 10, b: 10

Seguint el flux d'execució del programa obtenim:



34.12. Modificadors d'accés per una classe

Quan es declara una classe es pot especificar des de on es pot accedir a la classe en qüestió, això es fa amb un especificador d'accés a la declaració de la classe.

Només hi ha dos especificadors d'accés que operen a nivell de classe:

Deixar l'especificador d'accés en blanc

- Especifica el nivell d'accés a **package**.
- Només es pot accedir a la classe des d'una classe del mateix package.

public

- Es pot accedir a la classe des de qualsevol classe de qualsevol aplicació.

Exemple:

```

class Classe1 {
    ...
}

```

```
public class Classe2 {  
    ...  
}
```



Per accedir a una classe pública des d'una classe d'un package extern cal especificar el **fully qualified name** de la classe, és a dir, cal indicar el package de la classe a on s'accedeix.



En Java cada classe **publica** ha de residir en un fitxer **.java** independent amb el mateix nom que la classe.

Si la classe té àmbit de **paquet**, **pot residir al mateix fitxer** que una altra classe pública. Tingueu en compte, però, que en general no és una bona pràctica d'estil.

El següent codi és correcte:

Fitxer Classe.java.

```
public class Classe {  
    ...  
}  
  
class Classe2 {  
    // Es pot declarar la Classe2 dins de Classe.java  
    // sempre i quant Classe2 no sigui publica  
    ...  
}
```

Exemple:

```
package paquet1;  
public class Classe1 {  
    ...  
}  
  
package paquet2;  
public class Classe2 {  
    paquet1.Classe1 o = new paquet1.Classe1();  
}
```

34.13. Revisió de la clàusula import

Per accedir a una classe pública d'un package extern tenim tres opcions:

Utilitzar el *fqn* de la classe

Per exemple, podríem instanciar un objecte **Scanner** fent referència al nom complet de la classe:

```
java.util.Scanner in;
in = new java.util.Scanner(System.in);
```

Importar la declaració de la classe

La sintaxi general és:

```
import fqn-tipus;
```

Per exemple:

```
import paquet1.Classe11;
import paquet1.Classe12;
import paquet2.Classe21;
import paquet3.Classe31;
import paquet3.Classe32;
```

Habilitar la importació "on demand" de les classes del package extern

Per importar més d'un tipus d'un mateix package sense especificar els tipus específicament es pot fer mitjançant un mecanisme anomenat importació sota demanda.

```
import nom-paquet.*;
```

La construcció anterior importa tots els tipus de *nom-paquet*



La importació sota demanda només importa els tipus definits dins el package, no importa els tipus definits dins d'un altre package contingut dins del primer. Per exemple:

```
// La classe C1 resideix al paquet p1 i la
// classe C2 resideix al paquet p2 que està a la
// vegada dins de p1
import p1.*; // No importa el tipus C2
import p1.p2.*; // Ara sí;
```

34.14. Declaració de mètodes en una classe

Els mètodes dins de les classes defineixen el comportament dels objectes de la classe o el comportament de la propia classe.

```
modifiers tipus-de-retorn nom-del-mètode (llista-de-
paràmetres) clausula-throws {
    cos-del-mètode
}
```

La sintaxi referent a la declaració d'un mètode és molt similar a la de la declaració d'una funció.

Signatura d'un mètode

La **signatura d'un mètode** identifica el mètode de forma unívoca. És la combinació de:

- El seu nom
- El nombre de paràmetres
- El tipus dels paràmetres
- L'ordre dels paràmetres

Una classe **no pot tenir més d'un mètode amb la mateixa signatura**.

34.15. Variables locals

Una variable declarada dins d'un mètode, un constructor o un bloc s'anomena una **variable local**.

Una variable local declarada dins d'un mètode, un constructor o un bloc només existeix mentre el mètode, constructor o bloc està en execució. No es pot utilitzar fora del mètode, constructor o bloc on està declarada.

Particularitats de les variables locals:

- Les variables locals no s'inicialitzen per defecte, a diferència de les variables d'instància o de classe.
- Com a conseqüència de l'anterior punt no es pot accedir a una variable local fins que se li ha assignat un valor.
- Poden ser declarades a qualsevol punt del cos d'un mètode. Es considera una bona pràctica declarar les variables prop del seu ús.
- **Amaguuen** el nom de les variables de classe o d'instància amb el mateix nom, aquesta característica es coneix amb el nom de **shadowing**.

```
class Classe1 {
    int test = 10;

    void m1() {
        int test = 20;
        System.out.println(test); // 20
    }
}
```

34.16. Mètodes d'una instància i mètodes d'una classe

De forma similar al que veiem amb les variables d'instància i les variables de classe podem definir mètodes pertanyents a una instància determinada o a la pròpia classe, amb l'ajuda de la paraula **static**.

El modificador **static** s'utilitza per definir un mètode de classe, l'absència de modificador defineix un mètode de les instàncies.

```
// Un mètode estàtic
static void Test() {
    ...
}

// Un mètode no estàtic
```

```
void Test() {  
    ...  
}
```



Un mètode estàtic es pot cridar des de qualsevol instància o des de la pròpia classe.



Es poden cridar un mètodes estàtics de la pròpia classe des de mètodes estàtics de la classe.

```
class C1 {  
    static void m1() {  
  
    }  
  
    static void m2() {  
        m1(); // Correcte  
    }  
}
```

No es poden cridar mètodes d'instància de la pròpia classe des de mètodes estàtics de la classe.

```
class C1 {  
    void m1() {  
        ...  
    }  
  
    static void m2() {  
        m1(); // Error, no sabem de quina de les  
        instàncies es crida el mètode  
    }  
}
```

34.17. Invocar un mètode

Invocar un mètode consisteix en **executar el codi del cos d'un mètode**.

Els mètodes de les classes i els mètodes dels objectes es criden diferent.

Per invocar un **mètode d'un objecte** es crida el mètode a partir d'una referència a l'objecte i amb l'ajuda de l'operador .

```
referència.nom-del-mètode(paràmetres);
```

```
Punt2D p = new Punt2D(1, 2);
Punt2D resultat = p.multiplicar(3);
```

Per invocar un **mètode d'una classe** es crida el mètode a partir de la pròpia classe o a partir d'una referència a un objecte de la classe.

```
classe.nom-del-mètode(paràmetres);
```

o bé

```
referència.nom-del-mètode(paràmetres);
```

```
Punt2D p = new Punt2D(1, 2);
Punt2D resultat = Punt2D.multiplicar(p, 3);
```

o bé

```
Punt2D p = new Punt2D(1, 2);
Punt2D resultat = p.multiplicar(p, 3);
```

34.18. Constants

Un cas clar on s'utilitza la clàusula **static** és alhora de definir constants dins d'una classe.

En aquest cas no té sentit tenir una constant per cadascuna de les instàncies ja que en totes elles valdrà el mateix valor, per tant es declararà com a **static**.

Per exemple:

```
public class MathUtils {
    public static final double PI = 3.1416;
    ....
```

```
{}
```

34.19. Revisió del mètode main

La declaració del mètode main utilitza dos modificadors, **public** i **static**.

El modificador **public** fa que la funció sigui accessible des de qualsevol punt de l'aplicació sempre i quant la classe on estigui encapsulat el mètode sigui accessible.

El modificador **static** fa que no faci falta tenir una instància de la classe per a cridar el mètode main, cosa que té sentit ja la JVM no sap com crear un objecte de la classe que conté el mètode main i per tant necessita una manera estàndard de cridar-lo.

El següent exemple és correcte.

```
public class TestMain {
    public static void main(String[] args) {
        TestMain tm = new TestMain();
        tm.start();
    }

    public void start() {
    }
}
```

34.20. Paraula clau this

La paraula clau **this** és una referència a la instància actual de la classe, **només es pot utilitzar en el context d'una instància**.

No es pot utilitzar en el context d'una classe ja que **this** indica la instància actual i en el context d'una classe no existeix cap instància.

Exemples:

1. Moltes vegades no és necessari utilitzar la referència **this**, per exemple:

```
public class TestThis {
```

```

int valor;

public void setValor(int a) {
    valor = a;
}
}

```

2. **this** és útil per a distingir un camp d'una variable local amb el mateix nom.

```

public class TestThis {
    int a = 10;

    public void metode() {
        int a = 5;
        System.out.println(a); // Fa referència a la variable local
        System.out.println(this.a); // Fa referència a la variable de
l'objecte actual
    }
}

```

3. De forma similar a l'exemple anterior.

```

public class TestThis {
    int a = 10;

    public void metode(int a) {
        System.out.println(a); // Fa referència a la variable local
        System.out.println(this.a); // Fa referència a la variable de
l'objecte actual
    }
}

```

4. Els mètodes poden tornar objectes, en particular poden tornar l'objecte on resideixen.

```

public class Persona {
    int edat;
    ...
    public Persona mesVell(Persona p ) {
        return (p.edat > edat ? p : this);
    }
}

```

34.21. Modificadors d'accés pels membres d'una classe

Els modificadors d'accés associats als membres d'una classe determinen quines àrees del programa poden accedir-hi.



Quan es parla dels mebres d'una classe es fa referència als **camps** i als **mètodes** de la classe.

Taula 34.1. Especificadors d'accés pels membres d'una classe

Accés	Especificador	Accessibilitat
privat	private	Només a la mateixa classe
package	(sense especificador)	Dins del mateix package
protegit	protected	Mateix package o descendent a qualsevol package
public	public	Des de qualsevol lloc

34.22. Representació de classes en UML

El **Unified Modelling Language** és el llenguatge de modelat de sistemes més conegut i més utilitzat actualment, està mantingut pel [Object Management Group¹](#).

És un llenguatge gràfic per visualitzar, especificar, construir i documentar un sistema. UML proporciona un estàndard per descriure els "plànols" d'un sistema incloent-hi aspectes conceptuals com ara processos i funcions del sistema i aspectes concrets com expressions de llenguatges de programació i esquemes de bases de dades.

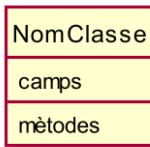
UML disposa d'una sèrie de diagrames diferents que permeten mostrar diferents aspectes de les entitats representades.

En particular estem interessats en un diagrama anomenat **Diagrama de classes UML** que representa una vista **estàtica** de l'estructura d'un sistema mostrant les seves classes amb els seus atributs, operacions i les relacions entre objectes.

¹ <http://www.omg.org>

Una classe en UML es representa amb un rectangle dividit en tres seccions horitzontals.

A la **primera secció** apareix el nom de la classe, a la **segona** els camps de la classe i a la **tercera** els mètodes de la classe.



Els especificadors d'accés dels camps i els mètodes s'indiquen afegint un símbol davant del seu nom:

Especificador d'accés	Representació en UML
public	+
private	-
protected	#
sense especificador	~

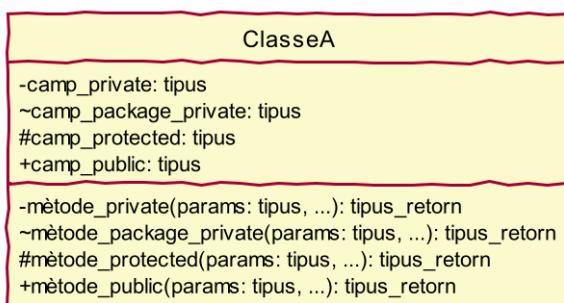


Figura 34.1. Representació d'una classe en UML dibuixada a mà

Algunes aplicacions modifiquen els símbols estàndards per fer-los més atractius:

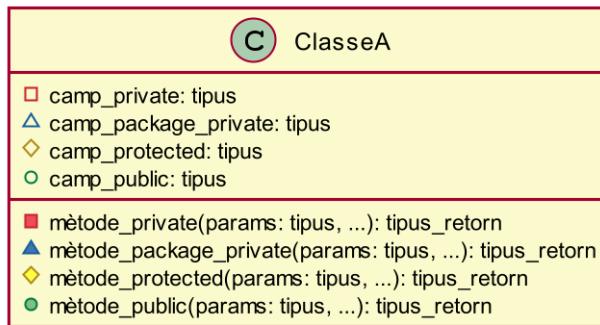


Figura 34.2. Representació d'una classe UML en PlantUML

Els camps i els mètodes mostrats en els dos exemples anteriors representen **camps i mètodes dels objectes**. Els camps i mètodes de la **Classe** es representen **subratllats**.

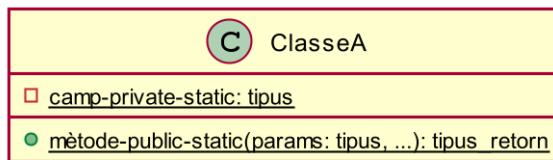


Figura 34.3. Representació d'un camp estàtic en UML

34.23. Primer principi de la programació orientada a objectes, l'encapsulació i l'amagament de la informació

Un dels tres principis fonamentals de la programació orientada a objectes és l'encapsulació que es pot definir de la següent manera:

Encapsulació

Vincular les dades amb el codi que les manipula, és a dir, no permetre l'accés a les dades d'una classe directament sinó a través d'alguns dels seus mètodes.

Per exemple, considerem el subsistema de direcció d'un vehicle. Es tracta d'un sistema complex format per una sèrie de components que treballen acobladament i que permeten moure el vehicle en la direcció escollida. No obstant, pel món exterior, existeix una sola interfície disponible per a tot el sistema, tota la seva complexitat s'amaga darrera d'aquesta interfície. A més

Primer principi de la programació orientada a objectes, l'encapsulació i l'amagament de la informació
a més, el sistema de direcció d'un vehicle és en sí un sistema complet i independent, no afecta el funcionament d'altres mecanismes.

De forma similar, el concepte d'encapsulació es pot aplicar al codi. El codi ben encapsulat té les següents característiques:

1. Tothom sap com accedir-hi.
2. És usable independentment dels detalls d'implementació.
3. No genera "efectes secundaris" a la resta del codi.

El concepte d'encapsulació està relacionat amb el concepte d'**abstracció**.

Abstracció

És el procés que determina quines dades i funcionalitats son rellevants i quines no, i per tant quins mètodes d'una classe han de ser visibles i quins no.



Els conceptes d'abstracció i d'encapsulació són complementaris, l'abstracció és centra en el comportament observable de l'objecte i l'encapsulació en la implementació que dona peu a aquest comportament.

Això ens porta al **principi d'amagament de la informació**.

Principi d'amagament de la informació

Els objectes es comuniquen amb altres objectes utilitzant **interfícies ben definides**.

Els objectes no **tenen permès** conèixer els detalls d'implementació dels altres objectes. Els detalls d'implementació estan **amagats o encapsulats** dintre de la classe.

El principi d'amagament de la informació facilita la **reutilització de la classe**.

34.24. L'encapsulació a la pràctica

El conjunt de tots els mètodes proporcionats per una classe junt amb la descripció del seu comportament s'anomena **interfície pública de la classe**.

La interfície pública d'una classe només està formada per mètodes, per tant una classe com la següent no està ben encapsulada ja que la seva interfície pública està formada per dos camps, no dos mètodes.

```
public class Alumne {  
    int edat;  
    String nom;  
}
```

El **principi de encapsulació obliga** que tots els camps de les classes en Java tinguin l'especificador d'accés *private*.

Si es vol proporcionar accés de lectura o escriptura a un determinat camp caldrà fer-ho a través d'un mètode implementat de manera específica.

Per exemple:

```
public class Alumne {  
    private int edat;  
    private String nom;  
  
    public int getEdat() {  
        return this.edat;  
    }  
  
    public void setEdat(int edat) {  
        this.edat = edat;  
    }  
  
    public String getNom() {  
        return this.nom;  
    }  
  
    public void setNom(String nom) {  
        this.nom = nom;  
    }  
}
```

En Java s'anomenen **setters** o **mutators** als mètodes que s'utilitzen per a modificar els camps privats d'una classe.

En Java s'anomenen **getters** o **accessors** als mètodes que s'utilitzen per a llegir els camps privats d'una classe.

En general s'anomenen **proprietats** d'un objecte a aquells valors representats internament per camps però accessibles a través de getters o setters.



Per conveni, en Java tots els getters s'anomenen `getNomCamp` i tots els setters s'anomenen `setNomCamp`.

34.25. Constructors

Un constructor és un bloc de codi que s'utilitza per inicialitzar un objecte immediatament després que l'objecte sigui creat.

La sintàxi general per a la declaració d'un constructor és:

```
modificadors nom-constructor (llista-paràmetres) throws llista-
excepcions {
    // Codi del constructor
}
```

modificadors

Els modificadors d'un constructor poden ser **public**, **private**, **protected** o **sense modificador**.

nom-constructor

Ha de coincidir amb el **nom** simple de la classe.

llista-paràmetres

Llista de paràmetres opcional seguint la sintaxi dels paràmetres d'entrada dels mètodes.

Per exemple:

```
public class Test {
    public Test() {
        // Codi del constructor
```

```
}
```



Un constructor no declara un tipus de retorn i per tant no és un mètode.

Per exemple, la següent classe no declara cap constructor explícitament:

```
public class Test {
    public void Test() {
        // No és un constructor
    }
}
```

El constructor d'una classe es crida amb l'ajuda de l'operador *new*, per exemple, per a cridar el constructor de la classe Test faríem:

Classe Test.

```
public class Test {
    public Test() {
        // Codi del constructor
    }
}
```

Crida al constructor.

```
Test t;
t = new Test(); // crida al constructor i torna una referència a
l'objecte creat
```

El següent exemple crea dos objectes gat:

```
public class Gat {
    public Gat() {
        System.out.println("Miau...");
    }
}
```

```
public class GatTest {
```

```

public static void main(String[] args) {
    // Creem un objecte Gat i no guardem la referència
    new Gat();

    // Creem un objecte Gat i emmagatzemem la referència
    Gat g = new Gat();
}
}

```

Miau...
Miau...

34.26. Sobrecàrrega de constructors

Una classe pot tenir més d'un constructor. En aquesta situació diem que el constructor està sobrecarregat.

Les regles per a la sobrecarrega dels constructors són similars a les regles per a la sobrecarrega dels mètodes, **si una classe té més d'un constructor, aquests s'han de diferenciar en el nombre, tipus o ordre dels paràmetres.**

Per exemple:

```

public class Gos {
    private String nom;
    private double preu;

    // Constructor 1
    public Gos() {
        this.nom = "Desconeugut";
        this.preu = 0.0;
    }

    // Constructor 2
    public Gos(String nom) {
        this.nom = nom;
        this.preu = 0.0;
    }

    // Constructor 3
    public Gos(String nom, double preu) {
        this.nom = nom;
    }
}

```

```

        this.preu = preu;
    }

    public String detalls() {
        return "Nom: " + this.name + "\nPreu: " + this.preu";
    }
}

```

34.27. Cridar un constructor des d'un altre constructor

A vegades és útil cridar un constructor des d'un altre constructor, per exemple:

El següent exemple no compila.

```

public class Gos {
    private String nom;
    private double preu;

    // Constructor 1
    public Gos() {
        Gos("Desconegut", 0.0);
    }

    // Constructor 2
    public Gos(String nom) {
        Gos(nom, 0.0);
    }

    // Constructor 3
    public Gos(String nom, double preu) {
        this.nom = nom;
        this.preu = preu;
    }

    public String detalls() {
        return "Nom: " + this.name + "\nPreu: " + this.preu";
    }
}

```

El codi anterior no compila, Java proporciona una sintaxi específica per a cridar un constructor des d'un altre constructors amb l'ajuda de la paraula clau *this*.

El codi anterior quedaría:

```

public class Gos {
    private String nom;
    private double preu;

    // Constructor 1
    public Gos() {
        this("Desconeugut", 0.0);
    }

    // Constructor 2
    public Gos(String nom) {
        this(nom, 0.0);
    }

    // Constructor 3
    public Gos(String nom, double preu) {
        this.nom = nom;
        this.preu = preu;
    }

    public String details() {
        return "Nom: " + this.name + "\nPreu: " + this.preu;
    }
}

```

Existeixen **dues regles importants** alhora de cridar constructors des d'altres constructors:

1. Si un constructor crida a un altre constructor, ha de ser la primera cosa que faci dins el cos del constructor.

El següent codi gener un error de compilació:

```

public class Test {
    int valor;

    public Test() {
        valor = 10; // Error!!
        this(valor);
    }

    public Test(int valor) {
        this.valor = valor;
    }
}

```

```

    }
}
```

2. Un constructor no es pot cridar a si mateix ja que resultaria en una crida recursiva. El següent codi gener un error de compilació:

```

public class Test {
    int valor;

public Test() {
    this(); // Error
}

public Test(int valor) {
    this.valor = valor;
}
}
```

34.28. Constructor per defecte

Totes les classes necessiten almenys un constructor, si no, no es podrien crear instàncies d'aquella classe.

Per tant si una classe **no declara explícitament un constructor** el compilador de Java **genera un constructor per a la classe** de forma automàtica.

Aquest constructor s'anomena **constructor per defecte**.



Si una classe ja implementa un constructor, el compilador de Java **no** afegeix un constructor per defecte.

En una classe de primer nivell, el constructor per defecte serà **public o a nivell de paquet** en funció de l'especificador d'accés de la classe i la seva signatura serà la d'un constructor sense paràmetres.

Per exemple:

Codi font de la classe	Versió compilada de la classe
<pre>public class Test {</pre>	<pre>public class Test { public Test() {</pre>

Representació de les relacions
entre classes en UML

	}
class Test { }	public class Test { Test() } }
public class Test { public Test() { } }	public class Test { public Test() { } }
public class Test { public Test(int) { } }	public class Test { public Test(int x) { } }

34.29. Representació de les relacions entre classes en UML

Dependència

La relació de *dependència* és la menys formal de totes. Representa que existeix algun tipus de dependència entre les dues classes.

És una relació ambigua i convé no abusar-ne.

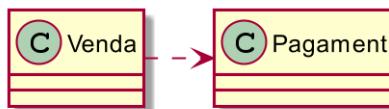


Figura 34.4. Representació de la relació de dependència en UML

En Java aquesta relació indicaria que s'ha hagut d'importar la classe perquè algú mètode la utilitza com a paràmetre. Per exemple:

```

imports tpv.pagaments.Pagament;
. . .
class Venda {
    public void tancarVenda(Pagament p) {
        . . .
    }
}
  
```

Associació

La relació d'associació defineix un tipus de dependència **molt més forta** que la relació de *dependència*.

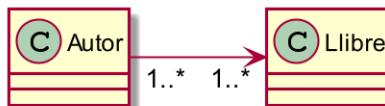


Figura 34.5. Representació de la relació de associació en UML

En Java aquesta relació sol indicar que la classe origen manté una referència a la classe destí en un camp de la classe. Per exemple:

```
class Autor {  
    Private Llibre llibre;  
  
    public Llibre getLlibre() {  
        return llibre;  
    }  
    . . . .  
}
```

Agregació

A vegades un objecte d'una classe conté un objecte d'una altra classe, indicant una relació **tot-part**.

Si a més a més l'objecte que fa el rol de **part** pot existir independentment de l'objecte que té el rol de **tot**. Aquest tipus de relació s'anomena **agregació**.

Per exemple, la relació entre **Biblioteca** i **Llibre** es podria modelar com:



Figura 34.6. Representació de la relació de agregació en UML

```
public class Llibre {  
}  
  
public class Biblioteca {  
    private Llibre llibre;
```

```
public Persona(Llibre llibre) {  
    thisllibre = llibre;  
}  
}
```

Composició

La **composició** és un cas especial d'agregació en el que l'objecte continent controla el cicle de vida de l'objecte contingut.

La diferència principal amb una relació d'agregació és que amb la composició l'objecte contingut no pot existir si no existeix l'objecte continent.

Per exemple, la relació entre **Biblioteca** i el seu **Catàleg** de llibres es pot modelar com una composició, l'existència del Catàleg està supeditada a l'existència de la Biblioteca.

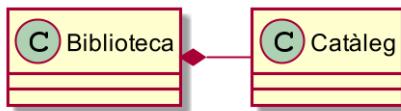


Figura 34.7. Representació de la relació de composició en UML

Java proporciona un mecanisme que es pot utilitzar per representar la composició. Es basa en un tipus de classes anomenades **inner class** que queden fora de l'àmbit d'aquest curs.

Exemple composició sense inner class.

```
class Cataleg {  
}  
  
class Biblioteca {  
  
    private final Cataleg cataleg; // Impedim canviar el contingut  
    de catàleg  
  
    public Biblioteca() {  
        cataleg = new Cataleg(); // Es crea l'objecte Cataleg  
        internament  
    }  
}
```

```
// public Catalog getCatalog(), no es pot afegir un getter de
catalag perquè una classe externa e podria quedar una referència i
el catalag sobrevisuria a la mort de la Biblioteca.
}
```

34.30. Representació de les relacions entre classes en UML - Exemple

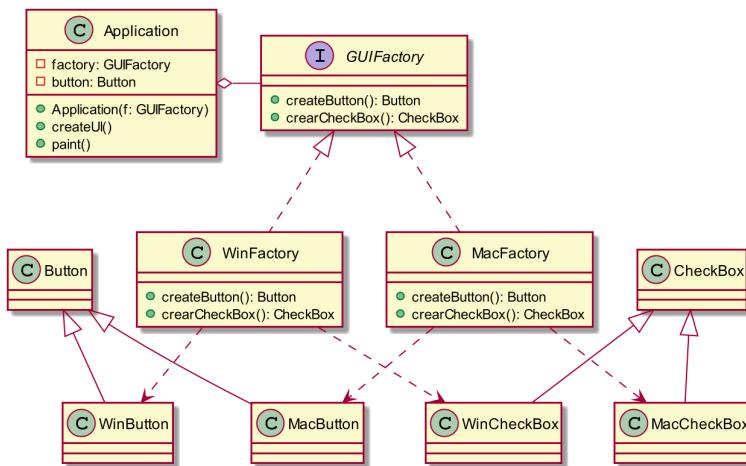
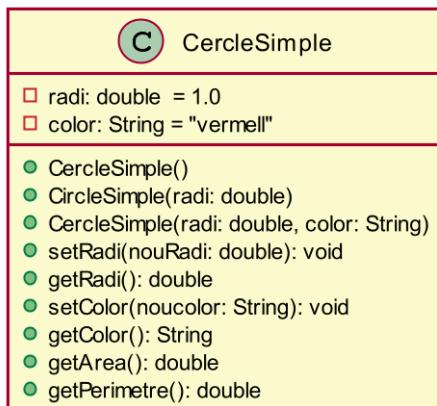


Figura 34.8. Exemple d'un diagrama de classes amb diferents relacions entre classes

34.31. Exemples

34.31.1. CercleSimple



```
package cerclesimple;

public class TestCercleSimple {

    public static void main(String[] args) {
        CercleSimple cercle1 = new CercleSimple();
        System.out.println("L'àrea del cercle de radi " +
            cercle1.getRadi() + " és " + cercle1.getArea());

        CercleSimple cercle2 = new CercleSimple(25);
        System.out.println("L'àrea del cercle de radi " +
            cercle2.getRadi() + " és " + cercle2.getArea());

        CercleSimple cercle3 = new CercleSimple();
        System.out.println("El color del cercle de radi " +
            cercle3.getRadi() + " és " + cercle3.getColor());

        cercle2.setColor("blau");
        cercle2.setRadi(100);
        System.out.println("el perímetre del cercle de color " +
            cercle2.getColor() + " és " + cercle2.getPerimetre());
    }
}

class CercleSimple {
    public static final double DEFAULT_RADI = 1.0;
    public static final String DEFAULT_COLOR = "vermell";

    private double radi;
    private String color;

    public CercleSimple() {
        this(DEFAULT_RADI, DEFAULT_COLOR);
    }

    public CercleSimple(double radi) {
        this(radi, DEFAULT_COLOR);
    }

    public CercleSimple(double radi, String color) {
        this.radi = radi;
        this.color = color;
    }
}
```

```

public void setRadi(double radi) {
    this.radi = radi;
}

public double getRadi() {
    return radi;
}

public void setColor(String color) {
    this.color = color;
}

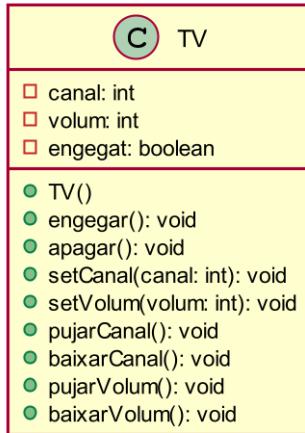
public String getColor() {
    return color;
}

public double getArea() {
    return radi * radi * Math.PI;
}

public double getPerimetre() {
    return 2.0 * radi * Math.PI;
}
}

```

34.31.2. TV



```
package tv;
```

```
public class TestTV {

    public static void main(String[] args) {
        TV tv1 = new TV();
        tv1.engegar();
        tv1.setCanal(30);
        tv1.setVolum(3);

        TV tv2 = new TV();
        tv2.engegar();
        tv2.pujarCanal();
        tv2.pujarCanal();
        tv2.pujarVolum();

        System.out.println("Canal tv1: " + tv1.getCanal()
                           + " Volum: " + tv1.getVolum());
        System.out.println("Canal tv2: " + tv2.getCanal()
                           + " Volum: " + tv2.getVolum());
    }
}

class TV {

    private int canal = 1;
    private int volum = 1;
    private boolean engegar = false;

    public TV() {
    }

    public void engegar() {
        engegar = true;
    }

    public void apagar() {
        engegar = false;
    }

    public void setCanal(int canal) {
        if (engegar && canal >= 1 && canal <= 120) {
            this.canal = canal;
        }
    }
}
```

```
public int getCanal() {
    return canal;
}

public void setVolum(int volum) {
    if (engegar && volum >= 1 && volum <= 7) {
        this.volum = volum;
    }
}

public int getVolum() {
    return volum;
}

public void pujarCanal() {
    if (engegar && canal < 120) {
        canal++;
    }
}

public void baixarCanal() {
    if (engegar && canal > 1) {
        canal--;
    }
}

public void pujarVolum() {
    if (engegar && volum < 7) {
        volum++;
    }
}

public void baixarVolum() {
    if (engegar && volum > 1) {
        volum--;
    }
}
```

```
Canal tv1: 30 Volum: 3
Canal tv2: 3 Volum: 2
```


35

Principis de disseny de classes

Ara que ja sanem què és una classe val la pena anomenar alguns principis de disseny generals que apliquen a l'hora de crear-ne de noves.

35.1. Principi de Cohesió

Una classe hauria de descriure una sola entitat, les seves operacions haurien de d'encaixar entre elles per donar suport a un propòsit coherent.

Una sola entitat amb més d'una responsabilitat es pot fragmentar en diverses classes per separar les responsabilitats, per exemple, les classes **String**, **StringBuilder** i **StringBuffer** treballen totes amb cadenes però tenen diferents responsabilitats.

La classe **String** encapsula cadenes immutables, la classe **StringBuilder** gestiona cadenes mutables i la classe **StringBuffer** és similar a la classe **StringBuilder** però conté mètodes sincronitzats per modificar cadenes.

35.2. Principi de Consistència

Consisteix en seguir l'estil estàndard de programar en Java i les convencions de noms. Trian noms informatius per les classes, els atributs i els mètodes.

Per exemple, una convenció popular consisteix en situar els atributs abans dels constructors i els constructors abans que els mètodes.

Per exemple, és una mala pràctica triar diferents noms per operacions similars, si **length()** retorna la mida d'una cadena per la classe **String** té sentit que s'utilitzi el mateix nom per retornar la mida de la cadena a les classes **StringBuffer** i **StringBuilder**.

35.3. Principi d'encapsulació

Una classe ha d'utilitzar el modificador **private** per amagar les seves dades de l'accés directe dels clients.

Proporcionar un *getter* només en el cas que es vulgui que el camp tingui permisos de lectura i un *setter* en el cas que es consideri que el camp ha de tenir permisos d'escriptura.

35.4. Principi de Claredat

La cohesió i l'encapsulació són bones guies per obtenir claredat en el disseny. Adicionalment, una classe ha de tenir un contracte clar que sigui fàcil d'explicar i fàcil d'entendre.

Els usuaris utilitzaran les classes en diferents combinacions, en diferent ordre i en diferents entorns. Les classes han d'estar dissenyades de manera que **no** imposin restriccions sobre què o quan pot un usuari utilitzar els diferents mètodes de la classe.

Els mètodes han de ser el més intuïtius possibles sense causar confusions.

36

La classe Object (Primera part)

La classe object es troba al paquet **java.lang.Object**. Totes les classes que s'inclouen a les llibreries de Java i totes aquelles que es poden crear estenen directa o indirectament la classe Object.

Totes les classes de Java són **subclasses** de Object i Object és **superclasse** de totes les classes.

La classe Object no té cap superclass, és la primera classe d'una jerarquia de classes de la qual pengen totes les demés classes.

36.1. Totes les instàncies són un objecte

Una variable de tipus Object pot mantenir una referència a qualsevol objecte de qualsevol classe.

Per tant totes les següents assignacions són vàlides:

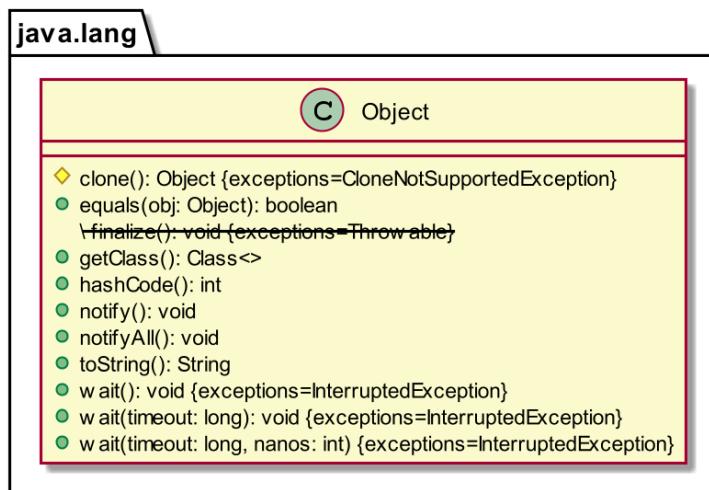
```
Object o;  
  
o = null;  
o = new Object();  
o = new Scanner(System.in);  
o = new Persona();
```

36.2. Els mètodes de la classe Object

La classe **Object** té 9 mètodes. Com que tota instància és un objecte, qualsevol instància de qualsevol classe tindrà aquests 9 mètodes.

Els mètodes **getClass()**, **notify()**, **notifyAll()** i **wait()** tenen una implementació **final** i per tant la seva implementació és la mateixa per a qualsevol objecte.

Els mètodes **toString()**, **equals()**, **hashCode()**, **clone()** i **finalize()** tenen una implementació per defecte a la classe object però altres classes **poden variar aquesta implementació**.



clone(): Object

S'utilitza per fer una còpia d'un objecte.

equals(obj: Object): boolean

S'utilitza per a comparar dos objectes per igualtat.

finalize(): void

Es cridat per la GC abans que un objecte sigui destruït.

getClass(): Class

Retorna una referència a l'objecte classe del objecte.

hashCode(): int

Retorna un codi de "hash" d'un objecte

notify(): void

Avisa a un fil d'execució a la cua d'espera del objecte.

notifyAll(): void

Avisa a tots els fils d'execució a la cua d'espera del objecte.

toString(): String

Retorna una representació en cadena d'un objecte. Normalment s'utilitza amb propòsits de depuració.

wait(): void, wait(timeout: long): void, wait(timeout: long, nanos: int)

Fa que el thread d'execució esperi a la cua d'espera de l'objecte.

36.3. Quina és la classe d'un objecte?

En Java cada objecte és una instància d'una classe, les classes es defineixen en codi font que posteriorment es compilàrà en un format binari que serà carregat a la JVM amb l'ajuda d'un objecte anomenat **class loader**.

Un "class loader", que és una instància de la classe **java.lang.ClassLoader**, llegirà la representació binària d'una classe des d'un sistema de fitxers o des de la xarxa per exemple, després crearà un objecte d'una classe anomenada **java.lang.Class** que representarà la representació binària de la classe dins de la JVM.

És a dir, podem pensar que una classe dins de la màquina virtual de Java és una instància del codi font de la classe compilada. Les classes carregades a la màquina virtual de Java són a l'hora objectes de la classe **java.lang.Class**.

El mètode **getClass()** de la classe **Object** retorna una referència a un objecte de tipus **Class**

Per exemple:

```
Persona p = new Persona();
Class classePersona = p.getClass();
```

Per defecte la definició d'una classe es **carrega una sola vegada a la JVM**, per tant cridar el mètode **getClass()** per a diferents objectes de la mateixa classe hauria de retornar una referència al mateix objecte **Class**.

Per exemple:

```

Persona p1 = new Persona();
Persona p2 = new Persona();

Class classePersona1 = p1.getClass();
Class classePersona2 = p2.getClass();

System.out.println(classePersona1 == classePersona2); // true

```

36.4. Comparar objectes per saber si són iguals

Tots els objectes són diferents entre ells, tots tenen una **identitat única**, per tant des del punt de vista de Java no té sentit plantejar-se si dos objectes són iguals.

El que si té sentit és detectar si dues referències a un objecte són iguals, és a dir, si dues variables "apunten" al mateix objecte. Això és el que comprova l'operador (`==`).

A vegades ens interessa considerar que dos objectes són iguals en base a altres paràmetres, per exemple si dos objectes tenen el mateix estat. Si es vol comparar dos objectes de manera diferent a la proposada per Java cal reimplementar el mètode `equals()`.



Per defecte, el mètode `equals()` de la classe Objecte està implementat de la següent manera:

```

public boolean equals(Object obj) {
    return (this == obj);
}

```

És a dir, per defecte està implementat de la mateixa manera que l'operador `==`.

Per exemple, considerem la classe següent:

```

class Punt {
    private double x;
    private double y;

    Punt(double x, double y) {
        this.x = x;
    }
}

```

```
    this.y = y;
}
}
```

I comprovem el funcionament per defecte del mètode **equals()**.

```
Punt p1 = new Punt(1.0, 1.0);
Punt p2 = new Punt(1.0, 1.0);
Punt p3 = p1;

System.out.println(p1 == p2); // false
System.out.println(p1.equals(p2)); // false

System.out.println(p1 == p3); // true
System.out.println(p1.equals(p3)); // true
```

Modifiquem el comportament del mètode **equals** per que consideri que dos objectes Punt són iguals si tenen les mateixes coordenades:

```
class Punt {
    private double x;
    private double y;

    Punt(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public boolean equals(Object obj) {

        if (this == obj) {
            return true;
        }

        if(obj == null) {
            return false;
        }

        if (this.getClass() != obj.getClass()) {
            return false;
        }

        Punt aux = (Punt) obj;
        if (aux.x != this.x || aux.y != this.y) {
            return false;
        }
        return true;
    }
}
```

```
    if (aux.x == this.x && aux.y == this.y) {  
        return true;  
    } else {  
        return false;  
    }  
}
```

I comprovem el funcionament de la nova classe:

```
Punt p1 = new Punt(1.0, 1.0);  
Punt p2 = new Punt(1.0, 1.0);  
Punt p3 = p1;  
  
System.out.println(p1 == p2); // false  
System.out.println(p1.equals(p2)); // true  
  
System.out.println(p1 == p3); // true  
System.out.println(p1.equals(p3)); // true
```

36.4.1. Especificacions alhora de reimplementar el mètode equals()

Quan es reimplementa el mètode **equals()** cal seguir les següents especificacions si es vol garantir el correcte funcionament de la classe amb la resta de les classes Java.

El mètode **equals()** ha de complir:

Comparació amb el mateix tipus d'objecte

Garantir que la classe des objectes que es comparen és la mateixa. Això és garantir que **this.getClass() == obj.getClass()**.

Reflexivitat

x.equals(x) ha de retornar **true**.

Simetria

x.equals(y) ha de donar el mateix que **y.equals(x)**.

Transitivitat

Si **x.equals(y) == true** i **y.equals(z) == true** aleshores **x.equals(z) == true**.

Consistència

- Si **x.equals(y) == true**, **x.equals(y) == true** mentre no es modifiquin els valors de **x** o de **y**.
- Si **x.equals(y) == false**, **x.equals(y) == false** mentre no es modifiquin els valors de **x** o de **y**.

Comparació amb null

o.equals(null) ha de ser false per a tot objecte.

Relació amb hashCode()

Si **x.equals(y)** aleshores **x.hashCode() == y.hashCode()**.

36.5. Calcular el "hash code" d'un objecte

Un "hash code" és un **valor enter** calculat a partir d'un **bloc d'informació** utilitzant un algoritme. La funció que calcula un "hash code" a partir d'un bloc d'informació s'anomena **funció de hash**.

Calcular un "hash code" és un procés d'una sola direcció, no és possible obtenir la informació original des del valor calculat.

36.5.1. Per què els objectes mantenen un "hash" code?

Quan es treballa amb col·leccions amb molts elements, fer cerques utilitzant el mètode **equals** pot ser extraordinàriament lent en funció de la complexitat del mètode.



El **hashcode** s'utilitza per determinar ràpidament la **no igualtat** entre dos objectes ja que **dos objectes iguals no poden tenir diferents codis de hash**.

Quan un objecte s'emmagatzema en una col·lecció basada en hash, en primer lloc es calcula un codi hash del objecte que s'utilitza per indexar l'objecte dins de la col·lecció. Aquest procés millora la eficiència de la col·lecció alhora de buscar elements.

Com que l'espai de resultats d'una funció de hash és menor que l'espai de combinacions del bloc d'informació inicial, varis blocs d'informació poden donar el mateix valor de hash. Per què una funció de hash sigui eficient cal que la distribució de resultats sigui uniforme dins el seu rang de valors.

Per exemple:

Una pèssima funció de hash.

```
public int calcularHash(String s) {  
    return 0;
```

Una millor aproximació.

```
public int calcularHash(String s) {  
    int suma;  
    for(int i = 0; i <s.length; i++) {  
        suma += s.charAt(i);  
    }  
  
    return suma % 24;  
}
```

En el cas d'un objecte, la informació que s'utilitza per a calcular un codi de hash és el propi estat de l'objecte, és a dir, el contingut dels camps de l'objecte.

36.5.2. Quan cal reimplementar el mètode *hashCode()*

La classe Object proporciona una implementació per defecte del mètode **hashCode()** basat en el valor de les referències emmagatzemades al la pila, per tant qualsevol classe ja té una implementació per defecte d'aquest mètode.

El problema ve quan es reimplementa el mètode **equals()**, dos objectes considerats iguals haurien de tenir el mateix hash.

La implementació per defecte de **equals()** és compatible amb la implementació per defecte de **hashCode()**, **dos objectes iguals són per defecte, el mateix, i per tant tenen la mateixa referència i per tant el mateix codi de hash**.

Si reimplementem el mètode **equals()** caldrà reimplementar el mètode **hashCode()** en consonància.

36.5.3. Exemple

Un **HashSet** és una col·lecció que representa un conjunt. Admet elements no ordenats i **no admets repetits**.

Si no redefinim el mètode **hashCode()** de la classe **Punt**, el següent codi no funciona correctament, té errors lògics.

```
import java.util.HashSet;

public class GetHashTest {

    public static void main(String[] args) {
        HashSet<Punt> set = new HashSet<>();

        Punt p1 = new Punt(1,1);
        Punt p2 = new Punt(1,1);
        Punt p3 = new Punt(1,1);

        set.add(p1);
        set.add(p2);
        set.add(p3);

        System.out.println("Nombre d'elements del conjunt: " +
set.size());
    }
}
```

Nombre d'elements del conjunt: 3 ①

① Hauria de donar un sol element!!

Redefinint el mètode **hashCode()** de **Punt** obtenim el resultat correcte.

```
public class Punt {

    private double x;
    private double y;

    Punt(double x, double y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public boolean equals(Object obj) {

        if (this == obj) {
```

```

        return true;
    }

    if (obj == null) {
        return false;
    }

    if (this.getClass() != obj.getClass()) {
        return false;
    }

    Punt aux = (Punt) obj;

    if (aux.x == this.x && aux.y == this.y) {
        return true;
    } else {
        return false;
    }
}

@Override
public int hashCode() { ①
    return (int) (this.x + this.y);
}
}

```

- ① Redefinim **hashCode()**.

Nombre d'elements del conjunt: 3

36.6. Representació d'un objecte amb una cadena

A vegades és útil, sobretot quan es depura una aplicació, mostrar un objecte en un format cadena.

En general ens interessa mostrar tot l'estat de l'objecte en un format llegible.

El mètode **toString()** de la classe Object proporciona aquest tipus de comportament.

La implementació per defecte del mètode **toString()** és mostrar el **FQN** de la classe seguit del "hash code" de l'objecte, per exemple:

```
Object o = new Object();
System.out.println(o.toString());
```

```
java.lang.Object@41ae8
```

Per exemple:

```
class Punt {
    ...
    public String toString() {
        return "(" + this.x + ", " + this.y + ")";
    }
    ...
}
```



Java crida automàticament al mètode **toString()** quan s'utilitza l'operador de concatenació de cadenes o quan es crida a **System.out.print** directament passant un objecte com a paràmetre.

36.7. Exemple - Una classe completa

```
public class Punt {

    private double x;
    private double y;

    Punt(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double getX() {
        return x;
    }

    public void setX(double x) {
        this.x = x;
    }

    public double getY() {
```

```
    return y;
}

public void setY(double y) {
    this.y = y;
}

@Override
public boolean equals(Object obj) {

    if (this == obj) {
        return true;
    }

    if (obj == null) {
        return false;
    }

    if (this.getClass() != obj.getClass()) {
        return false;
    }

    Punt aux = (Punt) obj;

    if (aux.x == this.x && aux.y == this.y) {
        return true;
    } else {
        return false;
    }
}

@Override
public int hashCode() {
    return (int) (this.x + this.y);
}

@Override
public String toString() {
    return String.format("(%.1f, %.1f)", x, y);
}
```

36.8. Finalitzar un Objecte - Deprecated JDK11

La JVM gestiona una tasca de baixa prioritat anomenada **recol·lector de brossa** que elimina definitivament de la memòria tots aquells objectes que no estan referenciatos.

El recol·lector de brossa permet l'execució de codi de neteja abans que un objecte sigui destruït, això o fa cridant el mètode **finalize()** de la classe Object declarat de la següent manera:

```
protected void finalize() throws Throwable {
}
```

La implementació del mètode **finalize()** de la classe Object no fa res, cal sobreescrivir el mètode en una classe derivada.

El mètode **finalize()** serà cridat pel recol·lector de brossa abans d'eliminar l'objecte de la memòria. Per exemple:

```
class Finalize {
    private int x;
    public Finalize(int x) {
        this.x = x;
    }

    @Override
    public void finalize() {
        System.out.println("Finalizing " + this.x);
    }
}

public class FinalizeTest {
    public static void main(String[] args) {

        for(int i = 0; i < 20000; i++) {
            new Finalize(i);
        }
    }
}
```

Consideracions important respecte el mètode **finalize()**:

- El recol·lector de brossa crida a **finalize()** **una sola vegada**.
- La crida al mètode **finalize()** no implica que l'objecte es destruirà just després de la finalització del mètode.
- El recol·lector de brossa **no és determinista** això implica que:
 - No es pot determinar l'ordre de destrucció dels objectes i per tant l'ordre en que es cridaran els diferents mètodes **finalize()**.
 - No es pot determinar el moment en que el recol·lector de brossa s'activarà. Ni tan sols es pot determinar si s'activarà o no.

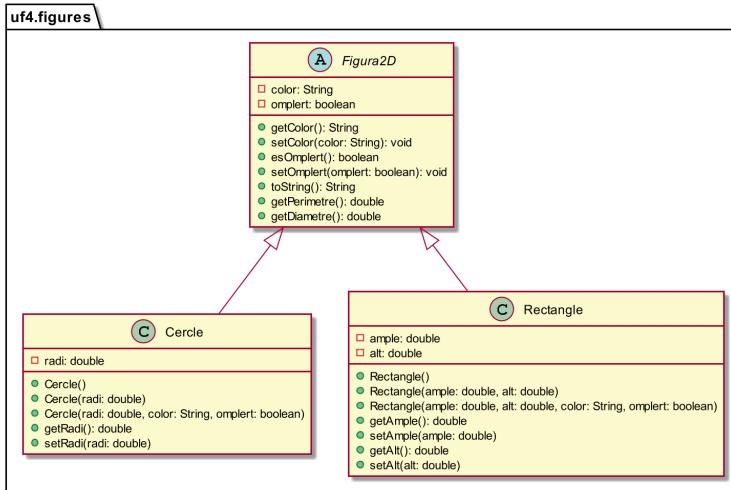
Degut a les consideracions anteriors **no es recomana** dependre del mètode **finalize()** per implementar la lògica de neteja dels objectes.

Herència

L'herència és una de les bases de la programació orientada a objectes. Permet crear una nova classe reutilitzant codi d'una classe existent. La nova classe s'anomena **subclasse** i la existent **superclasse**. Es diu que la subclasse hereta de la superclasse. Una superclasse també rep el nom de **classe base** o **classe pare**. Una subclasse també es coneix amb el nom de **classe derivada** o **classe filla**.

37.1. Superclasses i subclasses

L'herència permet definir una classe general i estendre-la posteriorment a una classe més especialitzada.



Alguns punts importants sobre l'herència:

- Contràriament a la interpretació convencional, **una subclasse no és un subconjunt de la seva superclasse**, al revés, en general una subclasse conté més informació i mètodes que la classe d'on deriva.
- Els **camps privats** d'una superclasse no són accessibles des de fora de la classe, com a conseqüència no es poden utilitzar directament des d'una subclasse. No obstant s'hi pot accedir a través de mètodes get/set si estan definits a la superclasse.
- No totes les relacions **és-un** es modelen correctament utilitzant l'herència. A vegades costa veure la manera correcte de fer-ho.

Quina és la millor alternativa?

Volem modelar la classe Quadrat.

- Un Quadrat és un Rectangle i per tant Quadrat hereta de Rectangle?
 - Un Rectangle és un cas particular de Quadrat, és un Quadrat amb els costats flexibles. I per tant Rectangle deriva de Quadrat?
 - Un Quadrat i un Rectangle no comparteixen una relació d'herència i per tant els fem derivar ambdós de Figura2D?
- L'herència modela relacions **és-un**. Procureu no derivar una classe simplement pel fet que es poden reutilitzar alguns mètodes. Per exemple, no té sentit heretar la classe Avio de la classe Ocell simplement perquè els dos volen.
 - Alguns llenguatges de programació permeten derivar una classe de més d'una classe pare. Aquesta capacitat s'anomena **herència múltiple**. Java no ho permet.

37.2. Representació de la relació d'herència en UML

La relació **és un** s'acostuma a modelar mitjançant **herència**.

Per exemple, la relació entre **Gos** i **Animal** es podria modelar com:

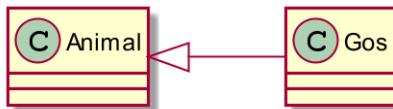


Figura 37.1. Relació d'herència

37.3. Paraula clau extends

La sintaxis que permet que una classe derivi d'una altre és la següent:

```

<<modificadorsClasse>> class <<nomSubclasse>> extends <<nomSuperclasse>>
{
}

```

Prenent les classes de l'exemple anterior:

```

public class Figura2D {
    . . .
}

public class Rectangle extends Figura2D {
    . . .
}

public class Cercle extends Figura2D {
    . . .
}

```

37.4. La classe Object és la superclasse per defecte

Si la classe no especifica que està heretant d'alguna altra, és a dir, no apareix la paraula clau **extends** a la seva declaració, **hereta automàtica i implícitament de la classe java.lang.Object**.

Per exemple, les següents declaracions són equivalents:

```

public class Punt {
    . . .
}

```

i

```
public class Punt extends java.lang.Object {  
    . . .  
}
```



Totes les classes que no deriven explícitament d'una altra mitjançant la paraula clau *extends*, deriven de la classe **java.lang.Object**.

37.5. Herència i la relació jeràrquica

La herència modela una relació "és un" entre la subclasse i la superclasse.

Una subclasse pot tenir les seves pròpies subclasses que poden tenir alhora les seves pròpies subclasses.

totes les classes en una relació d'herència formen una cadena amb una estructura d'arbre que rep el nom de **jerarquia de classes**.

Java permet una sola ascendència per classe, és a dir, **una classe pot tenir una única superclasse**. El contrari no és cert, una classe pot tenir tants descendents com calgui.

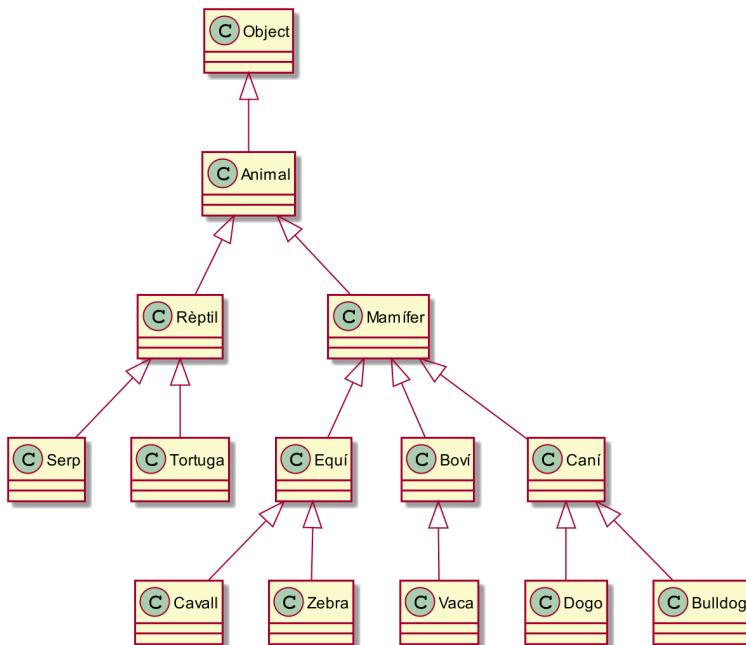


Figura 37.2. Exemple d'una jerarquia de classes

37.6. Quines coses són heretades per una subclasse?

Una subclasse **no ho hereta tot** de la seva superclasse. No obstant, una subclasse, **pot utilitzar tot el què** implementa la seva superclasse.

Una subclasse hereta:

- Tots els membres **no privats** de la classe pare i de classes ascendents.
- Els membres *sense especificador* si la subclasse està al mateix paquet que la superclasse.

Una subclasse **no hereta**:

- Els membres privats de la classe pare i de classes ascendents.
- Els constructors de la classe pare i de classes ascendents.
- Els inicialitzadors de la classe pare i de classes ascendents.
- Els membres *sense especificador* si la subclasse està dins **d'un paquet diferent** que la superclasse.

Recordem que existeixen quatre modificadors d'accés que es poden utilitzar tant pels membres d'una classe com pels constructors:

- public
- private
- protected
- sense especificador

Els especificadors anteriors determinen:

- En tots els casos: **qui pot accedir-hi.**
- En el cas dels membres: no dels constructors, **qui hereta.**

37.7. Herència múltiple

Alguns llenguatges de programació admeten derivar una classe de dos o més superclasses.

Aquest comportament pot portar problemes com al següent exemple:

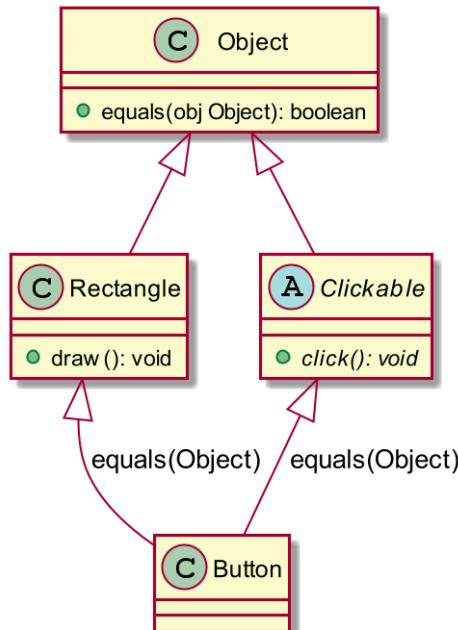


Figura 37.3. Problema del diamant amb l'herència múltiple

En l'exemple anterior no està definit quina de les dues implementacions del mètode **equals** hereta la classe Button. Aquest problema es coneix com el **problema del diamant**.

Per evitar aquest tipus de comportaments alguns llenguatges de programació, en particular **Java, C# i PHP**, no permeten **aquest tipus d'herència**.

37.8. Upcasting i Downcasting

Quan s'estableix una relació entre dues classes mitjançant herència la **subclasse és un tipus més específic que la superclasse**. Per exemple, un **Dogo** és un tipus particular d'**Animal**.

Per altra banda, la **superclasse comparteix un comportament** amb la subclasse. Per exemple, el comportament comú entre un **Dogo** i un **Bulldog** és ser un **Animal** o ser un **Caní**.

37.8.1. Upcasting

Des del punt de vista del codi, Java permet que un objecte d'una subclasse sigui tractat com un objecte d'una superclasse, aquest comportament respon al fet que un objecte d'una subclasse és en particular un objecte de la superclasse i per tant es pot posar en una variable de tipus superclasse.

Vegem-ho amb un exemple:

Exemples de "upcasting".

```
Dogo d = new Dogo();
Cani c = d; // Correcte, un Dogo és un Caní
Animal a = d; // Correcte, un Dogo és un Animal
a = c; // Correcte, un Caní és un Animal
Object o = d; // Correcte, tot objecte és un Object
o = c; // Correcte, tot objecte és un Object
o = a; // Correcte, tot objecte és un Object
```

37.8.2. Downcasting

El downcasting és just el contrari del upcasting, consisteix en especialitzar un objecte d'una classe en una de les seves subclasses.

Per exemple:

```
Dogo d = new Dogo();
Animal a = d; // correcte, upcasting.
Cani c = a; // downcasting, donarà un error ja que un Animal no és
necessàriament un Caní.
```

L'operació de downcasting **no és sempre possible**, per què es pugui realitzar cal que l'objecte inicial sigui del tipus de dades al que es vol assignar, en cas contrari es produirà un error.

Donat la naturalesa "perillosa" d'aquesta operació Java obliga al programador a certificar mitjançant un operador de "casting" que la conversió és possible.

Per exemple:

```
Dogo d = new Dogo();
Animal a = d; // correcte, upcasting.
Cani c = (Cani) a; // downcasting, ja no dona l'error perquè el
programador força la conversió i a més la variable a és efectivament un
Dogo.
```

En resum, per fer un downcasting cal que:

1. L'assignació sigui compatible.
2. Especificar l'assignació explícitament amb un operador de "casting".

37.9. Operador instanceof

Si es vol fer un downcasting verificant que l'assignació sigui compatible es pot utilitzar l'operador **instanceof**.

L'operador **instanceof** permet comprovar si una variable té una referència a determinada classe en temps d'execució.

La seva sintaxis és:

```
<<variableAmbUnaReferencia>> instanceof <<NomClasse>>
```

Alguns exemples:

```
class ClasseA {
```

Compte amb l'operador
instanceof i el mètode **equals**

```
    . . . .
}

class classeB extends ClasseA {
    . . . .
}

public class Main {
    public static void main(String args) {
        ClasseA a = new ClasseA();
        Object o = a;

        System.out.println(a instanceof Object);
        System.out.println(a instanceof ClasseA);
        System.out.println(o instanceof Object);
        System.out.println(o instanceof ClasseA);
        System.out.println(o instanceof ClasseB);
        System.out.println(a instanceof ClasseB);

        System.out.println(null instanceof Object); // la referència null
        sempre dona false amb qualsevol classe
    }
}
```

```
true
true
true
true
false
false
false
```

37.10. Compte amb l'operador **instanceof** i el mètode **equals**

Podem implementar el mètode **equals** utilitzant l'operador **instanceof** enlloc del mètode **getClass()** però podem tenir sorpreses.

Considerem el següent exemple:

Considerem que dues persones són iguals si tenen el mateix nom.

```
public class Persona {
    private String nom = "Desconegut";

    public void setNom(String nom) {
```

Compte amb l'operador
instanceof i el mètode **equals**

```
this.nom = nom;  
}  
  
public String getNom() {  
    return nom;  
}  
  
@overrides  
public boolean equals(Object obj) {  
    boolean esIgual = false;  
  
    if (obj instanceof Persona) {  
        persona e = (Persona)obj;  
        String n = e.getNom();  
        esIgual = n.equals(this.nom);  
    }  
    return esIgual;  
}  
  
// Cal sobreescrivir el mètode hashCode()!!  
}
```

Utilitzant **instanceof** permetem la comparació entre instàncies de diferents tipus sempre i quant deriven de la classe Persona, si s'utilitza **getClass()** només es permet la comparació entre objectes que siguin exactament instàncies de Persona.

Ara bé, l'exemple anterior **no és commutatiu**, és a dir:

```
class Persona {  
    // Considereu la implementació prèvia  
}  
class Alumne extends Persona {  
  
}  
  
public class Principal {  
  
    public static void main(String[] args) {  
        Persona p = new Persona();  
        Alumne a = p;  
  
        a.equals(p); // Retorna true  
        p.equals(a); // Retorna false
```

```
    }  
}
```

37.11. Sobreescritura de mètodes

Sobreescrivir un mètode consisteix en redefinir un mètode d'una classe en una subclasse.

Per exemple:

```
public class A {  
    public metode() {  
        System.out.print("A.metode()");  
    }  
}  
  
public class B extends A {  
    @overrides  
    public metode() {  
        System.out.print("B.metode()");  
    }  
}
```

En l'exemple anterior les instàncies de la classe A operaran amb la versió A de `metode()` i les instàncies de la classe B cridaran a la versió B de `metode()`.

Si per exemple, la classe C:

```
public class C extends B {  
    . . . .  
}
```

En aquest cas les instàncies de la classe C heretaran la versió de `metode()` de la classe B, no la versió de la classe A.

Un exemple:

```
package herencia;  
  
public class Herencia {
```

```
public static void main(String[] args) {
    A a = new A();
    B b = new B();
    C c = new C();
    D d = new D();
    E e = new E();

    a.metode();
    b.metode();
    c.metode();
    d.metode();
    e.metode();
}

}

class A {

    void metode() {
        System.out.println("A.metode()");
    }
}

class B extends A {

    void metode() {
        System.out.println("B.metode()");
    }
}

class C extends B {

}

class D extends C {

    void metode() {
        System.out.println("D.metode()");
    }
}

class E extends D {

}
```

```
A.metode()
B.metode()
B.metode()
D.metode()
D.metode()
```

Regles de sobreescritura:

1. La sobreescritura no s'aplica als mètodes **estàtics**.
2. La signatura (nom, tipus dels paràmetres, ordre dels paràmetres) del mètode ha de ser exactament la mateixa que la del mètode sobreescrit.

L'exemple següent **no és una sobreescritura**:

```
class A {
    public void metode() {

    }
}

class B extends A {
    public void metode(int a) { // Signatura diferent, no
        sobreescriu.

    }
}
```

3. El tipus del retorn d'un mètode que sobreescriu ha de ser:

- a. El mateix en el cas dels tipus primitius.
- b. Un tipus compatible en el cas dels tipus per referència.

```
class A {
    public Object metode() {

    }
}

class B extends A {
    public Point2D metode(int a) { //Point2D és un Object => OK
```

```
    }
}
```

4. L'**especificador d'accés** del mètode que sobreescrіu ha de ser el mateix o **menys restrictiu** que el del mètode sobreescrit.

37.12. Anotació @override

La anotació **@override** informa al compilador que l'element està sobreescrivint un element declarat en una superclasse.

Tot i que utilitzar l'anotació **@override** no és obligatori quan es sobreescrіu un mètode **ajuda a prevenir errors**.

Si un mètode marcat amb l'anotació **@override** falla alhora de sobreescrіure correctament un mètode d'una superclasse, el compilador genera un error.

37.13. Cridar un mètode sobreescrit des d'una classe filla, paraula clau super

A vegades serà necessari cridar la versió d'un mètode d'una classe pare sobreescrit en una classe filla.

Per fer-ho s'utilitza la paraula clau **super**.

```
public class ClassePare {
    public void metode(int valor) {
        . . .
    }
}
```

```
public class ClasseFilla extends ClassePare {
    @override
    public void metode(int valor) {
        . . .
    }

    public void calcul() {
        metode(); // Versió de ClassePare
        super.metode(); // Versió de la ClasseFilla
    }
}
```

{



Només es poden cridar mètodes de la classe immediatament superior. Per exemple la seüent sentència dona un error de compilació.

```
super.super.metode();
```

37.14. Upcasting i sobreescritura, "overriding"

Els mètodes s'ívoquen dinàmicament d'acord amb el **tipus de l'objecte** enlloc del **tipus de la variable que en manté una referència..**

Quan es fa un "upcasting" a un tipus diferent del de l'objecte **el tipus actual de l'objecte no canvia** només s'hi fa referència amb un tipus diferent.

El següent exemple il·lustra la situació:

```
package herencia;

public class Herencia {

    public static void main(String[] args) {
        A a;
        B b = new B();
        C c = new C();

        a = b;
        a.metode();
        a = c;
        a.metode();
    }
}

class A {

    void metode() {
        System.out.println("A.metode()");
    }
}
```

```
class B extends A {

    void metode() {
        System.out.println("B.metode()");
    }
}

class C extends B {

}
```

B.metode() // L'objecte és de tipus B i la referència de tipus A
 B.metode() // L'objecte és de tipus C i la referència de tipus A

37.15. Les variables en Java no es poden sobreescrivir

Una altra raó per la que no es bona idea treballar amb camps públics és que aquests no es poden sobreescrivir.

El següent exemple ho deixa clar:

```
class A {

    public char classe = 'A';
    void metode() {
        System.out.println("A.metode()");
    }
}

class B extends A {

    public char classe = 'B';
    void metode() {
        System.out.println("B.metode()");
    }
}

public class Main
{
    public static void main(String [] args)
    {
        A a = new A();
        B b = new B();
```

```
A aRef;
aRef = a;
System.out.println(aRef.classe);
aRef.metode();
aRef = b;
System.out.println(aRef.classe);
aRef.metode();
}
}
```

```
A
A.metode()
A
B.metode()
```

Per estalviar-nos problemes hauriem de reimplementar les classes de la següent manera:

```
class A {

    private char classe = 'A';

    char getClasse() {
        return classe;
    }

    void metode() {
        System.out.println("A.metode()");
    }
}

class B extends A {

    private char classe = 'B';

    char getClasse() {
        return classe;
    }

    void metode() {
        System.out.println("B.metode()");
    }
}
```

```
public class Main {

    public static void main(String[] args) {
        A a = new A();
        B b = new B();
        A aRef;
        aRef = a;
        System.out.println(aRef.getClasse());
        aRef.metode();
        aRef = b;
        System.out.println(aRef.getClasse());
        aRef.metode();
    }
}
```

```
A
A.metode()
B
B.metode()
```

37.16. Accés a mètodes sobreescrits

A vegades és necessari accedir a un mètode sobreescrit des d'una subclasse.

Una subclasse pot utilitzar la paraula reservada **super** per a cridar un mètode sobreescrit d'una superclasse.

Per exemple:

```
class A {
    void metode() {

    }
}

class B extends A {
    void metode() {

    }

    void proces() {
        metode(); // Estem cridant al mètode de la classe B
    }
}
```

```
super.metode() // Estem cridant al mètode de la classe A
}
}
```



Només es pot utilitzar **super** per a cridar mètodes de la classe immediatament superior.

37.17. Sobrecàrrega de mètodes, "overloading"

Quan hi ha més d'un mètode amb el mateix nom però **diferent signatura** a la mateixa classe es diu que el **mètode està sobrecarregat**.

Per tant, ni el **tipus de retorn**, ni l'**especificador d'accés**, ni el **nom dels paràmetres**, ni les clàusules **throws** tenen cap paper en la sobrecàrrega de mètodes.

Per exemple:

```
classe A {
    void metode() {
        . . .
    }

    int metode(int a) {
        . . .
    }

    void metode(float a, int b) {
        . . .
    }
}
```

37.18. L'herència i els constructors

Suposem la següent jerarquia de classes:

```
class A {
    protected int a1 = 1;
    protected int a2 = 2;
}
```

```

class B extends A {
    protected int b1 = 10;
}

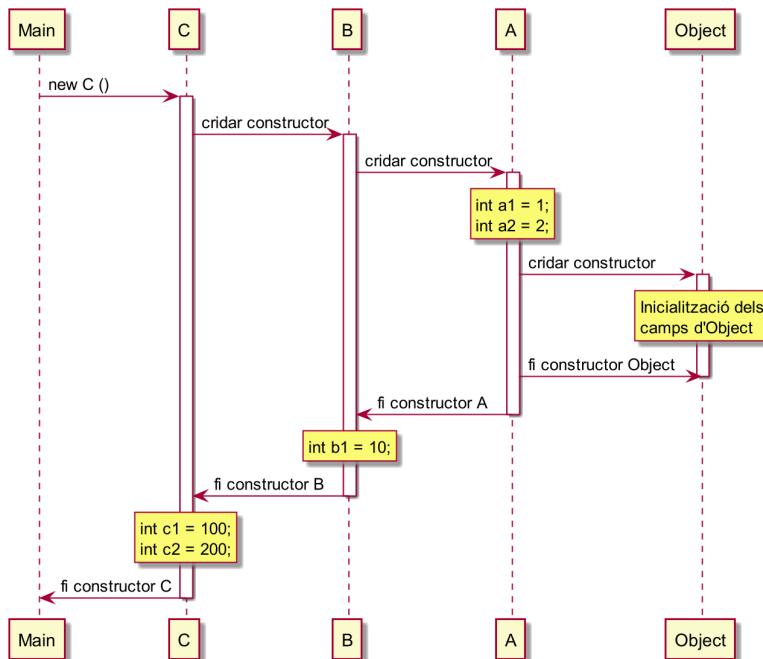
class C extends B {
    protected int c1 = 100;
    protected int c2 = 200;
}

public class Main {

    public static void main(String[] args) {
        C c = new C();
    }
}

```

Quan el mètode main crea una nova instància de la classe **C** el flux d'execució és el següent:



Té sentit, per crear una instància de la classe **C** cal reservar memòria per a tots els camps de l'objecte, com que l'objecte hereta de la classe **B**, de la classe **A** i

de la classe Object cal reservar també memòria pels camps heretats d'aquestes classes.

Per fer-ho cal cridar als constructors de cadascuna de les superclasses, recordeu que la missió dels constructors és inicialitzar l'estat dels objectes.

L'ordre de crida dels constructors és determinat per la jerarquia de les classes, la idea és que no té sentit tenir una instància de C si abans no es té una instància de B d'on derivar. I així successivament fins arribar a dalt de tot de la jerarquia.

Per defecte es criden els constructor per defecte de les superclasses, aquests poden ser implícits, autogenerats, o explícits, constructors sense paràmetres.

Per tant el següent exemple dona un error:

```
package constructors;

class A {

    protected int a1 = 1;
    protected int a2 = 2;
}

class B extends A {

    protected int b1;

    B(int b1) { // La classe B no té un constructor per defecte!!
        this.b1 = b1;
    }
}

class C extends B {

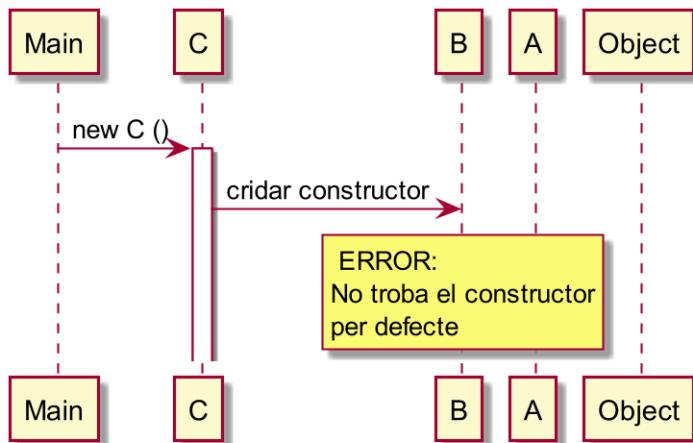
    protected int c1 = 100;
    protected int c2 = 200;
}

public class Main {

    public static void main(String[] args) {
        C c = new C();
    }
}
```

```
}
```

Com que la classe B té un constructor amb paràmetres, el constructor per defecte no existeix i per tant el recorregut de crides als constructors per defecte queda interromput.



Aquest problema té dues solucions:

1. Afegim un constructor sense paràmetres a la classe B:

```
package constructors;

class A {

    protected int a1 = 1;
    protected int a2 = 2;
}

class B extends A {

    protected int b1;

    B() { // Constructor sense paràmetres
        this(10);
    }

    B(int b1) {
        this.b1 = b1;
    }
}
```

```

    }

}

class C extends B {

    protected int c1 = 100;
    protected int c2 = 200;
}

public class Main {

    public static void main(String[] args) {
        C c = new C();
    }
}

```

2. Cridem explícitament al constructor amb paràmetres des del constructor de C. Per fer-ho utilitzarem la paraula clau **super()** i passarem els paràmetres adequats al constructor que volem cridar.

```

package constructors;

class A {

    protected int a1 = 1;
    protected int a2 = 2;
}

class B extends A {

    protected int b1;

    B(int b1) {
        this.b1 = b1;
    }
}

class C extends B {

    protected int c1 = 100;
    protected int c2 = 200;

    C() {

```

```

        super(10); // Cridem explícitament el constructor de B
    }
}

public class Main {

    public static void main(String[] args) {
        C c = new C();
    }
}

```



Cridar un constructor d'una superclasse des del constructor d'una subclasse ha de ser a la primera línia del constructor.

Té sentit si hem entès el funcionament de creació d'objectes de classes derivades.

Per exemple, el següent codi dona un error:

```

class A {
    private int a;

    A(int valor)
        this.a = a;
    }
}

class B extends A {
    private int b;
    B(int b) {
        this.b = b
        super(b); // No és la primera línia del
        constructor
    }
}

```

La versió correcte:

```

class A {
    private int a;
}

```

```

A(int valor)
    this.a = a;
}
}

class B extends A {
    private int b;
    B(int b) {
        super(B);
        this.b = b
    }
}

```

37.19. Amagar mètodes

TODO

37.20. Amagar camps

TODO

37.21. Deshabilitar l'herència

Es pot impedir que una classe es pugui derivar utilitzant la paraula clau **final** a la declaració de la classe.

Per exemple:

```

public final class ClasseA {
}

```

En aquest cas el següent codi donaria un error de compilació ja que no es pot derivar de ClasseA:

```

public class ClasseB extends ClasseA {
}

```

De forma similar, un mètode decorat amb la paraula clau **final** no podrà ser **sobreescrit** o **amagat** per una subclasse. Per exemple:

```

public class ClasseA {
}

```

```
public final void metode() {  
}  
}
```

37.22. Paraula clau final

La paraula clau final s'utilitza en Java en diversos contexts, en funció d'aquests la paraula final té diferents significats.

En general però, la idea que proporciona la paraula clau final és la de no permetre la modificació o el reemplaçament del valor original d'allò a que accompanya.

37.22.1. Variables final

Una variable declarada com a final només es pot inicialitzar una vegada. Això no significa que s'hagi de fer necessàriament en la declaració de la variable, sinó que es pot fer més endavant, quan ja hi hagi prou informació per saber quin valor ha de tenir.

37.22.2. Mètodes final

Un mètode es declara final quan no volem que es pugui sobreescrivir a una classe derivada. Això s'utilitza per prevenir errors quan una subclasse modifica un mètode que és crucial per al correcte funcionament intern de la classe.

En cas de dubte, és preferible **no declarar** com a final un mètode, ja que és complicat preveure totes les formes d'estendre la nostra classe i posar mètodes final pot limitar aquesta flexibilitat.

Hi ha pocs exemples de mètodes final a les API de Java, per exemple el mètode **getClass()** de la classe **Object**.

El resultat de sobreescrivir aquest mètode per retornar un tipus diferent d'objecte **Class** que el que li correspon a l'objecte seria desastrós.

37.22.3. Classes final

Una classe es declara *final* quan no es vol que es puguin crear classes derivades d'aquesta classe.

Per exemple, la classe **String** és final. Això s'ha fet així perquè la classe **String** és immutable, i no es vol que una classe derivada de **String** ho pugui canviar.

En el cas de **String**, si volem una cadena que es pugui modificar haurem d'utilitzar per exemple la classe **StringBuilder**.

El motiu pel qual s'ha dissenyat **String** de forma que sigui immutable té a veure amb els problemes que podem tenir amb els objectes mutables que emmagatzem com atributs en una classe, i pels quals es manté una referència externa.

Ho podem veure en el següent exemple:

```
public class Classe {  
    private Rectangle r;  
    private int area;  
  
    public Classe(Rectangle r) {  
        setRectangle(r);  
    }  
  
    @Override  
    public String toString() {  
        return r.toString();  
    }  
  
    public int getArea() {  
        return area;  
    }  
  
    public void setRectangle(Rectangle r) {  
        this.r = r;  
  
        area = r.width * r.height;  
    }  
}
```

En aquest exemple, estem calculant l'àrea del rectangle (**Rectangle** és una classe de les API de Java) en el mateix moment que el rebem, de manera que ens estalviem de calcular-la cada vegada que ens criden **getArea()**.

Com que **r** és una variable privada que només es pot modificar a través de **setRectangle**, podem estar segurs que mai no hi haurà una incongruència entre el rectangle que tenim i l'àrea calculada. **O no?**

Imaginem el següent programa:

```
public class Programa {
    public static void main(String[] args) {
        Rectangle rect = new Rectangle(2, 3, 4, 5); // x, y, amplada, alçada
        Classe cl = new Classe(rect);
        rect.width = 1000;

        System.out.println(cl);
        System.out.println("Àrea: "+cl.getArea());
    }
}
```

El resultat d'executar aquest codi és que ens mostrarà que el nostre rectangle de 1000x5 té una àrea de 20!

El problema ha estat el fet de mantenir una referència al rectangle intern de *cl*, que ens ha permet modificar un atribut privat.

Si *Rectangle* hagués estat immutable això no hagués pogut passar mai, perquè no s'hagués pogut executar la sentència *rect.width = 1000;*

Com que *String* s'usa de forma tant intensiva, les probabilitats de caure en aquest tipus d'errors augmenten, i per això el fer-la immutable i final és una bona mesura de seguretat.

I com arreglem el problema que tenim a l'exemple?

Doncs fent una còpia profunda del rectangle en comptes d'assignar una referència:

```
public void setRectangle(Rectangle r) {
    this.r = new Rectangle(r); // o this.r = r.clone();

    area = r.width * r.height;
}
```

37.23. Objectes immutables

Un **objecte immutable** és un objecte l'estat del qual no canvia un cop creat.

Un objecte immutable es pot veure des de dos punts de vista:

Extern

L'estat de l'objecte no es modificable des de l'exterior de l'objecte, no obstant l'objecte pot modificar el seu estat internament.

Intern

L'estat de l'objecte no canvia mai un cop establert per primera vegada.

En general quan es parla d'un objecte immutable es fa referència a la immutabilitat externa de l'objecte.

37.23.1. Quins avantatges tenen els objectes immutables

1. Els objectes immutables no generen problemes de sincronització i per tant funcionen bé en entorns amb més d'un fil d'execució.
2. Els objectes immutables soLEN ser bons candidats a claus per a les classes de tipus **Set** i de tipus **Map**.
3. Es podEN guardar en una "cache" ja que no canviaran mai.
4. Si un objecte immutable llençA una excepció mai quedara en un estat indeterminat o indesitjable.

37.23.2. Estratègies per a definir objectes immutables

1. No proporcionar "mutators".
2. Crear tots els atributs *final* i *private*.
3. No permetre que les subclasses sobreescriguin mètodes.
 - a. Fent la classe *final*
 - b. Fent el constructor *private* i proporcionant algun tipus de mètode factoria.
4. Si els atributs mantenen referències a objectes mutables no permetre que aquests objectes canviïn.
 - a. No proporcionar mètodes que modifiquin els objectes mutables.
 - b. No compartir referències als objectes mutables.
 - i. No emmagatzemar referències a objectes mutables externs passats al constructor

- ii. No retornar el valor dels atributs que mantenen referències a objectes mutables.

Exemple d'una classe immutable.

```
public class Valor {

    private final int referencia;

    public Valor(int valor) {
        this.referencia = valor;
    }

    public int getValor() {
        return referencia;
    }
}
```

Exemple d'una classe internament mutable i externament immutable.

```
public class Valor {

    private final int referencia;
    private int valorMig = Integer.MAX_VALUE;

    public Valor(int valor) {
        this.referencia = valor;
    }

    public int getValor() {
        return referencia;
    }

    public int getValorMig() {
        // Calcular el valor mig si no està calculat previament
        if (this.valorMig == Integer.MAX_VALUE) {
            // Un cop calculat per primera vegada s'emmagatzema el valor
            (cache)
            this.valorMig = this.referencia / 2;
        }
        return this.halfValue;
    }
}
```

L'exemple anterior treballa de forma similar a com treballa la classe **String** fent cache de les cadenes utilitzades per millorar el rendiment.

Una classe immutable mal implementada (de fet no és immutable).

```
public class Immutable {

    private final Valor valor;

    public Immutable(int valor) {
        this.valor = new Valor(valor);
    }

    public Valor getValor() {
        return new Valor(this.valor.getValor());
    }

    public int getInnerValor() {
        return valor.getValor();
    }
}
```

on la classe Valor és:

```
public class Valor {

    private int referencia;

    public Valor(int valor) {
        this.referencia = valor;
    }

    public int getValor() {
        return referencia;
    }

    public void setValor(int valor) {
        this.referencia = valor;
    }
}
```

Test de la classe anterior.

```
public class BadImmutableTest {
```

```

public static void main(String[] args) {
    Immutable i1 = new Immutable(101);
    int immutable = i1.getInnerValor();
    System.out.println("#1 valor = " + immutable);
    Valor valor = i1.getValor();
    holder.setValor(207);
    valor = i1.getValor();
    System.out.println("#2 valor = " + valor);
}
}

```

```

#1 valor = 101
#2 valor = 207

```

Clarament l'exemple anterior no és immutable, el problema és el mètode **getIntHolder()** que retorna una referència a un objecte mutable.

Versió modificada de l'exemple anterior, ara sí immutable.

```

public class Immutable {

    private final Valor valor;

    public Immutable(int valor) {
        this.valor = new Valor(valor);
    }

    public Valor getValor() {
        return new Valor(this.valor.getValor());
    }

    public int getInnerValor() {
        return valor.getValor();
    }
}

```

L'exemple anterior es correcte perquè el constructor admet com a paràmetre un valor de tipus primitiu i per tant quan l'assignem a una altra variable aquesta assignació és en realitat una copia del valor de la variable.

Però si volguessim tenir un constructor que admetés directament un objecte de tipus **IntHolder** haurien d'anar amb compte:

Versió modificada de l'exemple anterior amb el nou constructor mal implementada (no immutable).

```
public class Immutable {  
  
    private final Valor valor;  
  
    public Immutable(int valor) {  
        this.valor = new Valor(valor);  
    }  
  
    public Immutable(Valor valor) {  
        this.valor = valor;  
    }  
  
    public Valor getValor() {  
        return new Valor(this.valor.getValor());  
    }  
  
    public int getInnerValor() {  
        return valor.getValor();  
    }  
}
```

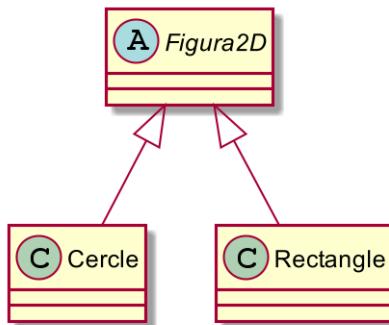
Versió modificada de l'exemple anterior amb el nou constructor, ara sí immutable.

```
public class Immutable {  
  
    private final Valor valor;  
  
    public Immutable(int valor) {  
        this.valor = new Valor(valor);  
    }  
  
    public Immutable(Valor valor) {  
        this.valor = new Valor(valor.getValor());  
    }  
  
    public Valor getValor() {  
        return new Valor(this.valor.getValor());  
    }  
  
    public int getInnerValor() {  
        return valor.getValor();  
    }  
}
```

}

37.24. Classes i mètodes abstractes

A vegades es vol crear una classe per representar un concepte enllot de per representar un objecte. Per exemple:



Crear una instància de la classe **Figura2D** no té massa sentit, té sentit crear alguna de les seves classes derivades.

La classe **Figura2D** s'ha creat per agrupar els mètodes i les propietats comuns a tots els seus descendents, no s'ha creat amb la idea de poder fer-ne instàncies directament.

Una classe **abstracta** és una classe de la que no es poden crear instàncies però de la que si que es poden derivar classes filles que poden ser **abstractes** o no.

Les classes abstractes es decoren amb la paraula clau **abstract**. Per exemple:

```

public abstract class Figura2D {
}
  
```

En aquest cas el següent codi donaria error:

```

Figura2D f2d = new Figura2D(); // ERROR, la classe Figura2D és
                                abstracta i no es poden crear-ne instàncies
  
```

De la mateixa manera que les classes poden ser **abstractes**, els mètodes també.

Un **mètode abstracte** és un mètode que **no té implementació** i per tant no es pot cridar directament.

Els mètodes abstractes estan dissenyats per a ser sobreescrits en les classes derivades i ser cridats des d'aquestes.

Per declarar un mètode abstracte es decora amb la paraula clau **abstract**. Per exemple:

```
public abstract class Figura2D {
    public abstract void pintar(int color); // Els mètodes abstractes no
    tenen cos i per tant no s'hi posen claus
}
```



Una classe amb un mètode abstracte ha d'estar declarada com a classe abstracta. Per exemple, el següent codi dona error:

```
public class Figura2D { // La classe té un mètode
    abstract i per tant s'ha de declarar com abstracte
    public abstract void pintar(int color);
}
```

Al revés **no** és cert, es pot tenir una classe abstracta sense cap mètode abstracte.

Quan es deriva d'una classe abstracta:

- **Cal implementar tots els mètodes abstractes** de la classe pare

```
abstract class A {
    abstract void metode();
}

abstract class B extends A {

}
```

- o bé, **cal declarar com a abstracte** la classe derivada.

```
abstract class A {
```

```

abstract void metode();
}

class B extends A {

    @Override
    void metode() {
        // Implementació de l mètode
    }
}

```

37.25. Composició per sobre de l'herència

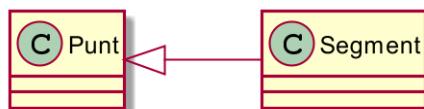
Tant la composició com l'herència permeten reutilitzar altres objectes dins d'una nova classe.

La composició s'utilitza generalment quan es volen les capacitats d'una classe existent dins de la nova classe, però no es vol la seva interfície. És a dir, s'inclou un objecte de manera que es pot utilitzar per implementar noves característiques, però l'usuari de la nova classe veu la interfície que s'ha definit allà, i no la de l'objecte inclòs.

Un dels principis del disseny orientat objecte assegura que cal “**afavorir la composició d'objectes per davant de l'herència de classes**”.

Per exemple:

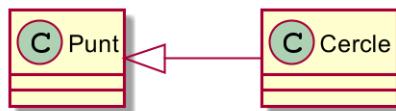
Considerar que la classe **Segment** hereta de la classe **Punt** és un error de disseny.



és millor modelar la relació per composició:



De la mateix manera podríem considerar que la classe Cercle deriva de la classe Punt. Tampoc és correcte, és millor utilitzar la composició.

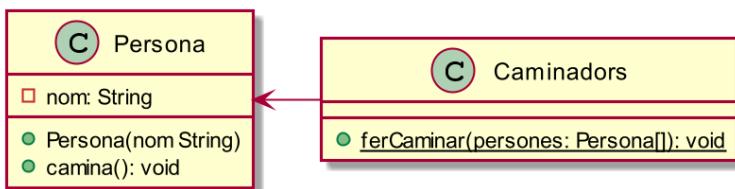


38

Interfícies

38.1. Tenim un problema...

Considerem la següent estructura de classes:



i el codi font associat:

```
public class Persona {
    private String nom;

    public Persona(String nom) {
        this.nom = nom;
    }

    public void camina() {
        System.out.println("En " + nom + ", que és una Persona, està
caminant.");
    }
}
```

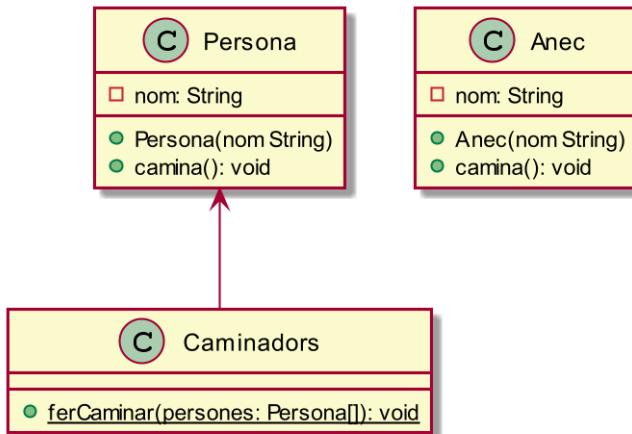
```
public class Caminadors {
    public static void ferCaminar(Persona[] personnes) {
```

```
for(int i = 0; i < persones.length; i++) {  
    persones[i].camina();  
}  
}  
}
```

```
public class CaminadorsTest {  
    public static void main(String[] args) {  
        Persona[] persones = new Persona[3];  
        persones[0] = new Persona("Joan");  
        persones[1] = new Persona("Josep");  
        persones[2] = new Persona("Jaume");  
  
        Caminadors.ferCaminar(persones);  
    }  
}
```

En Joan, que és una Persona, està caminant.
En Josep, que és una Persona, està caminant.
En Jaume, que és una Persona, està caminant.

Ara considerem la classe **Anec**, que també pot caminar, implementada de la següent manera:



```
public class Anec {  
    private String nom;  
  
    public Anec(String nom) {
```

```

    this.nom = nom;
}

public void camina() {
    System.out.println("En " + nom + ", que és un Ànec, està caminant.");
}
}

```

I ara, volem fer caminar els ànecs amb l'ajuda de la classe Caminadors.

El següent codi generarà un error de compilació:

```

public class CaminadorsTest {
    public static void main(String[] args) {
        Persona[] persones = new Persona[3];
        persones[0] = new Persona("Joan");
        persones[1] = new Persona("Josep");
        persones[2] = new Anec("Jaume"); // Error!!
        Caminadors.ferCaminar(persones);
    }
}

```

Considerem tres maneres de resoldre el problema anterior:

Primera solució

Canviar el codi de **ferCaminar(Persones[] persones)** de la classe Caminadors per què admeti una matriu d'**Object** enlloc d'una matriu de **Persona**.

```

public class Caminadors {
    public static void ferCaminar(Object[] objs) {
        for(int i = 0; i < persones.length; i++) {
            objs[i].camina(); // Error!!
        }
    }
}

```

El codi anterior segueix tenint un problema, el mètode **camina()** no forma part de la classe **Object** i per tant el codi no compila.

Podríem seguir endavant amb aquesta idea preguntant-li a la variable **objs** si té un mètode anomenat **camina()** i en aquest cas cridar-lo.

Això es pot fer amb una tècnica anomenada "**reflection**":

```
import java.lang.reflect.Method;
public class Caminadors {
    public static void ferCaminar(Object[] objs) {
        for(int i = 0; i < personnes.length; i++) {
            Method metodeCaminar = obtenirMetodeCaminar(obj[i]);
            if (metodeCaminar != null) {
                try {
                    metodeCaminar.invoke(objs[i]);
                } catch (Exception ex) {
                    ex.printStackTrace();
                }
            }
        }
    }

    public static Method obtenirMetodeCaminar(Object obj) {
        Class c = obj.getClass();
        Method metodeCaminar = null;
        try {
            metodeCaminar = c.getMethod("caminar");
            return metodeCaminar;
        } catch (NoSuchMethodException ex) {
            return null;
        }
    }
}
```

El problema d'aquesta solució es que es fonamenta en que **tots els objectes que caminen tenen un mètode anomenat exactament caminar()** cosa que no té per què ser així.

Segona solució

Afegir un mètode **ferCaminar(Anec[] anecs)** a la classe **Caminadors**.

```
public class Caminadors {
    public static void ferCaminar(Persona[] personnes) {
        for(int i = 0; i < personnes.length; i++) {
            personnes[i].camina();
        }
    }

    public static void ferCaminar(Anec[] anecs) {
```

```

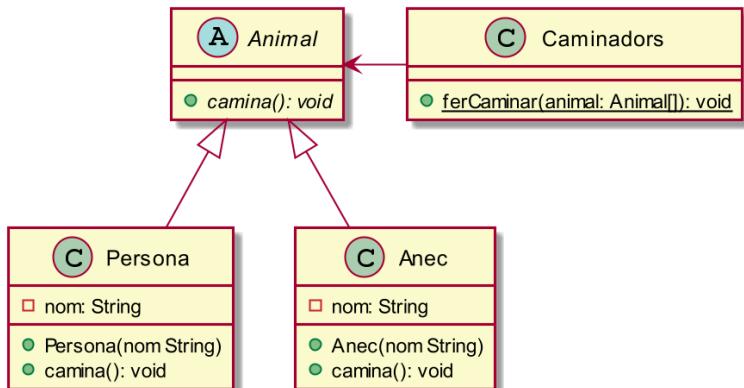
for(int i = 0; i < anecs.length; i++) {
    anecs[i].camina();
}
}
}

```

El problema principal d'aquesta solució és l'**extensibilitat**, a mesura que vagin apareixen noves classes amb el comportament "caminar" caldrà afegir nous mètodes a la classe **Caminadors**. A més, si es vol fer caminar un conjunt d'objectes variats que puguin caminar s'haurà de cridar independentment cadascun dels mètodes **ferCaminar()** implicats.

Tercera solució

Derivar la classe **Persona** i la classe **Anec** d'una classe **Animal** que declari el mètode **camina()** com a mètode abstracte.



La solució anterior té dos problemes:

- Què passa si necessitem una classe **Peix**? Dues opcions:
 - Peix** no la fem derivar d'**Animal**.
 - Tenim un **Peix** que hereta d'**Animal** i per tant implementa el mètode **camina()**, podríem llançar una excepció des d'aquest mètode indicant que un **Peix** no pot caminar.
- Què passa si apareix la classe **GosRobot** què no volem fer derivar de la classe **Animal** però si volem que implementi el mètode **camina()**.

38.2. Una solució mitjançant interfícies

Una interfície és una construcció que representa un contracte, la interfície defineix una sèrie de mètodes abstractes i una classe pot comprometre's a implementar tots els mètodes declarats en una interfície en particular.

Per exemple:

```
public interface Caminador {  
    void camina();  
}
```

Estem redactant un contracte que diu: Si vols ser un **Caminador** has d'implementar un mètode que es diu **camina()**.

Tant la classe Persona com la classe Anec volen ser **Caminador** i per tant implementen el contracte Caminador, això els obliga a implementar el mètode **void caminar()**.

```
public class Persona implements Caminador {  
    private String nom;  
  
    public Persona(String nom) {  
        this.nom = nom;  
    }  
  
    public void camina() {  
        System.out.println("En " + nom + ", que és una Persona, està  
        caminant.");  
    }  
}
```

```
public class Anec implements Caminador {  
    private String nom;  
  
    public Anec(String nom) {  
        this.nom = nom;  
    }  
  
    public void camina() {  
        System.out.println("En " + nom + ", que és un Ànec, està caminant.");  
    }  
}
```

```
}
```

Una interfície defineix un nou tipus de dades, per tant, el següent codi és correcte:

```
Caminador c;
```

Ara bé, **no** es pot crear un objecte d'una interfície. El següent codi dona error:

```
Caminador c;
c = new Caminador(); // Error!!
```

No obstant, una variable de tipus una interfície **pot fer referència a qualsevol objecte que implementi aquesta interfície**. El següent codi és correcte:

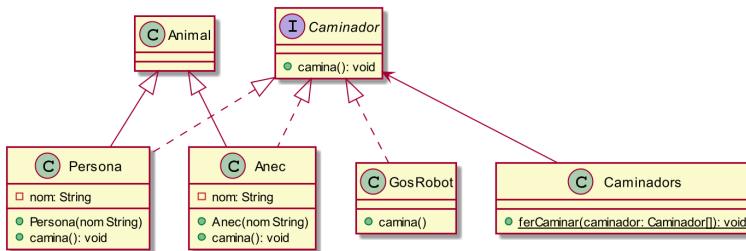
```
---
Caminador c1 = new Persona("Josep"); // Correcte, Persona implementa
Caminador
Caminador c2 = new Anec("Donald"); // Correcte, Anec implement Caminador
Caminador c2 = new Object(); // Error, Object no implementa Caminador
---
```

Per tant, podem modificar Caminador i CaminadorTest de la següent manera:

```
public class Caminadors {
    public static void ferCaminar(Caminador[] caminadors) {
        for(int i = 0; i < personnes.length; i++) {
            caminadors[i].camina();
        }
    }
}
```

```
public class CaminadorTest {
    public static void main(String[] args) {
        Caminador[] caminadors = new Caminador[3];
        caminadors[0] = new Persona("Joan");
        caminadors[1] = new Persona("Josep");
        caminadors[2] = new Anec("Donald");

        Caminadors.ferCaminar(caminadors);
```



38.3. Declarar una interfície

La paraula clau **interface** permet la declaració d'interfícies. La sintaxis utilitzada en Java és la següent:

```

modifiers interface nomInterficie {
    declaració-de-constants
    declaració-de-mètodes-abstracs
}
  
```

Els modificadors d'una interfície poden ser els mateixos que els d'una classe, **public** o **sense-especificador**

Per exemple:

```

public interface Dibuixable {
    // . . .
}
  
```

- Les interfícies només poden contenir **constants** o **declaracions de mètodes**.
- Una interfície pot estar buida.
- Els mètodes d'una interfície són **públics** per defecte i no cal especificar-ho explícitament amb la paraula clau **public**.
- Les interfícies són implícitament **abstractes** i no cal especificar-ho mitjançant la paraula clau **abstract**.
- Els camps declarats a una interfície són implícitament **publics**, **statics** i **finals**, les següents interfícies són equivalents:

```

public interface Pregunta {
  
```

```
public static final int SI = 0;  
public static final int NO = 1;  
}
```

```
public interface Pregunta {  
    int SI = 0;  
    int NO = 1;  
}
```

38.4. Mètodes estàtics a les interfícies

TODO

38.5. Mètodes per defecte a les interfícies

TODO

38.6. Declaració d'interfícies niuades

TODO

38.7. Una interfície defineix un nou tipus de dades

Una interfície defineix un nou tipus per referència.

Es pot utilitzar una variable de tipus una interfície a qualsevol lloc on tingui sentit utilitzar un tipus per referència: per declarar una variable, com a paràmetre en un mètode, com a retorn d'un mètode.

No obstant, **no** es pot crear un nou objecte de tipus una interfície.

38.8. Implementar una interfície

La paraula clau **implements** permet la implementació d'interfícies. La sintaxis utilitzada en Java és la següent:

```
modificadors class nomClasse implements nomInterficieSeparatsPerComes {  
}
```

Una classe que implementa una interfície està **obligada** a implementar **tots** els mètodes declarats a la interfície.

```
public class Pissarra {
    private List<Dibuixable> objectesDibuixables;

    public Pissarra(){
        objectesDibuixables = new ArrayList<Dibuixable>();
    }

    public void dibuixarTotsElsObjectes(){
        for(int i = 0; i < objectesDibuixables.size(); i++){
            objectesDibuixables[i].dibuixar();
        }
    }
}

public interface Dibuixable {
    void dibuixar();
}

public class Rectangle implements Dibuixable {
    public void dibuixar() {
        // Implementació de dibuixar
    }
}
```

38.9. Interfícies en UML



Figura 38.1. Representació d'una interfície en UML dibuixada a mà

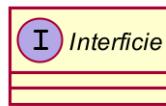


Figura 38.2. Representació d'una classe UML en PlantUML

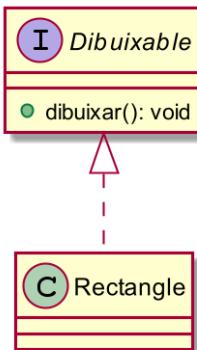


Figura 38.3. Implementació d'una interfície

38.10. Implementar múltiples interfícies

Una classe pot implementar més d'una interfície, per exemple:

```

public interface Camina {
    void camina();
}

public interface Neda {
    void neda();
}

public class Tortuga implements Neda, Camina {
    // implementació de la interficie Camina
    public void camina() {
        // ....
    }

    // Implementació de la interficie Neda
    public void neda() {
        // ....
    }

    // Implementació de la pròpia classe
    public void menja() {
        // ....
    }
}
  
```

38.11. Implementació parcial d'interfícies

Si recordem que una interfície és **abstracta** de forma implícita podem implementar parcialment una interfície si declarem la classe com a abstracta. Per exemple:

```
interface A {  
    void metode1();  
    void metode2();  
}  
  
public abstract class Classe implements A {  
    public void metode1() {  
        // implementació  
    }  
    // No cal implementar metode2() ja que al ser la classe abstracte pot  
    // ser implementat en una subclasse  
}
```

38.12. Les interfícies admeten herència (i herència múltiple)

Una interfície pot heretar d'una altra interfície. **A diferència de les classes una interfície pot heretar de més d'una interfície.**

Per exemple:

```
interface Escriptor {  
    void escriu();  
}  
  
interface Cantant {  
    void canta();  
}  
  
interface EscriptorDeLletres extends Escriptor, Cantant {  
}  
  
public class Lletrista implements EscriptorDeLletres {  
    @Override  
    public void escriu() {  
        // ...  
    }  
}
```

```
@Override  
public void canta() {  
    // ...  
}  
}
```

38.13. El Polimorfisme revisat

El polimorfisme fa referència a la provisió d'una interfície per entitats de diferents tipus i per tant a través de la mateixa interfície obtenim comportaments diferents en funció de l'objecte que hi ha al darrera.

En Java el un mateix objecte es pot veure a través de varies "vistes".

- A través de la vista proporcionada per la seva pròpia classe.
- A través de la vista proporcionada per alguna de les seves superclasses.
- A través de la vista proporcionada per alguna de les seves interfícies.

39

Principis de disseny a la programació orientada a objectes

39.1. Hem de sospitar que el nostre codi té problemes si...

És rígid

Petits canvis fan que el sistema s'hagi de reconstruir.

És fràgil

Canvis en un mòdul fan que deixin de funcionar correctament altres mòduls no relacionats.

És immòbil

Els components interns d'un mòdul no es poden extreure per ser reutilitzats en nous entorns.

És viscós

Testear i construir el sistema és difícil i porta molt de temps. És costós fer fins i tot un sol canvi, requereix fer canvis a més d'un lloc.

39.2. Principi de responsabilitat única

Una classe ha de tenir una, i només una, raó per canviar.

El principi de la responsabilitat única estableix que mai hi hauria d'haver més d'una raó per la qual una classe hagi de canviar. Això vol dir que cada classe ha de fer una sola feina.

Això no implica que les classes hagin de contenir un sol mètode o propietat, vol dir que tots els mètodes i propietats de la classe han d'estar relacionats amb una sola responsabilitat.

Per exemple, la següent classe no segueix el principi de la responsabilitat única:

```
class Treballador {  
    public Pay calcularSou() {...}  
    public void guardar() {...}  
    public String descriureTreballador() {...}  
}
```

Estem barrejant lògica de càlcul, lògica relacionada amb la base de dades i lògica relacionada amb la generació d'informes.

Tenir més d'una responsabilitat combinades en una sola classe fa que sigui **dificil modificar-ne una part sense haver de modificar-ne les altres**. A més genera classes més **difícils d'entendre i de comprovar**.

39.3. Principi d'obert-tancat

S'hauria de poder estendre el comportament d'una classe sense haver de modificar-la.

El principi "obert-tancat" determina que les classes han d'estar obertes per expansió però tancades per modificació.

Que les classes siguin **obertes per la expansió** significa que les classes s'han de dissenyar per què es pugui afegir nova funcionalitat a mesura que apareixen nous requeriments.

Que les classes siguin **tancades per modificació** significa que un cop desenvolupada una classe aquesta no s'hauria de modificar mai excepte per a corregir "bugs".

Aquest principi s'aconsegueix, en general, fent referència a **abstraccions** per resoldre les **dependències**, és a dir, utilitzant **interfícies i classes abstractes** en lloc de classes concretes. D'aquesta manera es pot afegir funcionalitat creant noves classes que implementin les interfícies.

Un efecte lateral de l'ús d'interfícies i classes abstractes és la reducció de les interdependències entre les diferents classes i l'augment de la flexibilitat.

Per exemple:

```
void pagar(Factura factura) {
    Import total = Import.zero;
    for (item : items) {
        total += item.getPreu();
        factura.addItem(item);
    }
    Pagament p = acceptCash(total);
    factura.addPagament(p);
}
```

Com es faria per afegir suport per pagar amb targeta de crèdit?

Una possibilitat és afegir un **if** similar al següent, cosa que trencaria el principi del "obert - tancat":

```
Pagament p;
if (credit) {
    p = acceptaCredit(total);
} else {
    p = acceptaCash(total);
}
factura.addPagament(p);
```

Una solució millor seria:

```
public interface MetodePagament {
    void acceptarPagament(Import total);
}

void pagar(Factura factura, MetodePagament pm) {
    Import total = Import.zero;
    for (item : items) {
        total += item.getPreu();
        factura.addItem(item);
    }
    Payment p = pm.acceptarPagament(total);
    factura.addPagament(p);
```

{}



El principi de "obert-tancat" és només útil quan els canvis de requeriments són predictibles. Si no ho són no s'hauria d'aplicar aquest principi, s'hauria de resoldre el problema utilitzant la solució més simple.

39.4. Principi de substitució de Liskov

Les classes derivades han de ser substituïbles per les seves classes pare.

El **principi de substitució de Liskov** s'aplica a les jerarquies de classes basades en herència.

El principi fa referència a que totes les subclasses haurien d'operar de la mateixa manera que ho fan les seves classes pare.

Les funcionalitats específiques de les subclasses poden ser diferents que les de les superclasses però han de ser coherents amb el comportament d'aquestes. És a dir, les subclasses no només han d'implementar els mètodes de les classes base sinó que a més s'han d'ajustar al seu comportament implícit.

En general si un subtipus d'un supertipus fa alguna cosa que el client del supertipus no espera segurament s'està produint una violació del principi de substitució de Liskov.

Alguns exemples del no compliment del d'aquest principi:

1. Una classe que llença una excepció que la superclasse no llença. Si tractem un objecte de la classe com a un objecte de la superclasse no tenim en compte ell llançament de la excepció.
2. Una classe Quadrat que deriva de la classe Rectangle. Si s'utilitza la classe Quadrat en un context on s'espera un Rectangle es poden produir efectes col·laterals ja que els costats d'un quadrat no poden modificar-se de manera independent.

```
public class Rectangle {
    private double height;
```

```

private double width;

public double area();

public void setHeight(double height);
public void setWidth(double width);
}

public class Square extends Rectangle {
    public void setHeight(double height) {
        super.setHeight(height);
        super.setWidth(height);
    }

    public void setWidth(double width) {
        setHeight(width);
    }
}

```

En general les violacions d'aquest principi produeixen comportaments no definits. Això més: Funcionament correcte en desenvolupament i funcionament incorrecte en producció.

39.5. Principi de la segregació d'interfícies

Fer interfícies ben granulades que siguin específiques pel client.

O no depenguis de coses que no necessites.

El principi de la segregació de les interfícies estableix que els clients no haurien de ser forçats a dependre de membres d'interfícies que no utilitzen.

Això porta a la creació de **múltiples** interfícies **petites** molt cohesionades que minimitzen les interdependències entre classes.

El següent exemple no segueix el principi de segregació de les interfícies.

Considerem un caixer automàtic amb la següent interfície:

```

public interface Messenger {
    demandarTargeta();
    notificarTargetaInvalida();
    demandarPin();
    notificarPinInvalid();
}

```

```

demanarCompte();
notificarNoHiHaDiners();
demanarConfirmacio();
notificarExtracte();
notificarBalanc();
}

```

Per seguir el principi de segregació d'interfícies hauríem de separar les diferents funcionalitats en diferents interfícies independents:

```

public interface LoginMessenger {
    demanarTargeta();
    notificarTargetaInvalida();
    demanarPin();
    notificarPinInvalid();
}

public interface WRetiradDinersMessenger {
    demanarCompte();
    notificarNoHiHaDiners();
    demanarConfirmacio();
}

public class CatalaMessenger implements LoginMessenger,
    WithdrawalMessenger {
    ...
}

```

39.6. Príncipi de la inversió de les dependències

Depèn de les abstraccions, no de les concrecions.

La idea d'aquest principi és que isolem les classes darrera una barrera formada per les abstraccions de les quals depenen.

Si tots els detalls queden darrera d'aquestes abstraccions quan es produeix un canvi en alguna d'elles les classes involucrades segueixen funcionant.

Això facilita l'adaptació als canvis i el testeig unitari de les diferents classes del sistema.

Per exemple, si un programa depèn del teclat i de la impressora per funcionar tindria sentit fer-ne abstraccions i treballar a través d'aquestes abstraccions. Així

si més endavant la impressora, per exemple, es substituís per una pantalla seria fàcil adequar el programa:

```
interface Reader {  
    char getchar();  
}  
  
interface Writer {  
    void setChar(char c);  
}  
  
class Impressora implements Writer {  
}  
  
class Teclat implements Reader {  
}
```

i treballariem a través de les interfícies.

```
Writer wr = new Impressora();  
Reader rd = new Teclat();
```

Part V. UF5 POO. Llibreries de classes fonamentals

Anybody who comes to you and says he has a perfect language
is either naïve or a salesman.

— Bjarne Stroustrup *Creador del llenguatge de programació C*

++

Sumari

40. Resultats d'aprenentatge i criteris d'avaluació	441
41. Continguts	443
42. Control d'excepcions	445
42.1. Llançament d'excepcions	445
42.2. Excepcions	447
42.3. Com es gestionen les "checked exception"	448
42.4. Com es gestionen les RuntimeException	449
42.5. Captura d'excepcions	450
42.6. Obtenir informació de les excepcions	452
42.7. Clàusula finally	453
42.8. Definir noves classes d'excepció	454
42.9. Construcció try -amb-recursos	456
42.10. Algunes recomanacions a l'hora de treballar amb excepcions ..	458
42.10.1. Llençar d'hora capturar tard	458
42.10.2. No emmascarar excepcions	459
42.10.3. Tenir en compte que algunes excepcions es poden produir al bloc finally	459
43. La classe ArrayList	463
43.1. java.util.ArrayList versió anterior a Java 5	463
43.2. java.util.ArrayList versió posterior a Java 5	469
44. Genèrics	471
44.1. Tipus genèrics	473
44.2. Implementació de la classe Box	473
44.3. La classe Box utilitzant genèrics	474
44.4. Noms de les <i>variables tipus</i>	475
44.5. Referències a classes genèriques	475
44.6. Classes genèriques amb més d'un paràmetre	475
44.6.1. Utilitzar classes genèriques sense <i>arguments tipus</i>	476
44.7. Tipus acotats (bounded types)	477
44.7.1. Múltiples tipus acotats	479
44.8. Arguments comodí	479
44.8.1. Primer intent de solució	480
44.8.2. Segon intent de solució	481
44.8.3. Tercer intent de solució	482

44.8.4. Codi d'exemple	482
44.9. Arguments comodí acotats	483
44.9.1. Codi d'exemple	487
44.10. Mètodes genèrics	490
44.11. Constructors genèrics	491
44.12. Interfícies genèriques	492
45. La interfície Comparable	495
46. La interfície Comparator	503
46.1. Comparator i Comparable per les mateixes classes	505
47. Col·leccions	507
47.1. Interfícies vs implementacions	507
47.2. Programar per la interfície, no per la implementació	508
47.3. Un exemple, una interfície, dues implementacions	509
47.3.1. La interfície	510
47.3.2. Les dues implementacions	510
47.3.3. Les proves	513
47.4. Interfícies	514
47.4.1. La interfície java.util.Collection	515
47.4.2. Interfície java.util.Queue (Cua)	516
47.4.3. Interfície java.util.Deque	517
47.4.4. Interfície java.util.List (Llista)	518
47.4.5. Interfície java.util.Set (Conjunt)	519
47.4.6. Interfície java.util.Map (Diccionari)	520
47.5. Implementacions	521
47.5.1. Implementacions de java.util.Queue	521
47.5.2. Implementacions de java.util.Deque	522
47.5.3. Implementacions de java.util.List	523
47.5.4. Implementacions de java.util.Set	523
47.5.5. Implementacions de java.util.Map	524
47.6. Recórrer col·leccions	527
47.7. Iteradors	527
47.8. Exemple d'implementació d'un iterador	532
47.8.1. Els iteradors de llistes: ListIterator	536
47.8.2. Inserir i extreure elements a través d'un iterador	536
47.8.3. Ús d'un iterador sense l'iterador: enhanced for	537
47.9. Algorismes: les classes Arrays i Collections	539

47.10. Configuració de JavaFX i Apache Netbeans amb Ant	541
47.10.1. Baixar i instal·lar la biblioteca	541
47.10.2. Afegir la biblioteca JavaFX globalment	541
47.10.3. Afegir JavaFX a un projecte concret	544
47.11. Configuració de JavaFX i Apache Netbeans amb Maven	547
48. JavaFX - Introducció	549
48.1. Introducció a les interfícies gràfiques en Java	549
48.2. Estructura bàsica d'un programa de JavaFX	550
48.3. Aplicacions amb més d'un Stage	552
48.4. Panells, Controls i Formes	553
48.5. Com s'utilitza un Pane amb un Control	555
48.6. La classe Pane	556
48.7. Com s'utilitza un Pane amb una Shape	557
48.8. Classe Color	559
48.9. Classe Font	559
48.10. Aplicar fulls d'estil CSS	561
48.10.1. Aplicar un estil directament	563
48.10.2. Crear un full d'estils i aplicar-lo a la escena	564
48.11. Classe Image i ImageView	568
48.12. Layouts	570
48.12.1. FlowPane	571
48.12.2. TextFlow	573
48.12.3. TilePane	576
48.12.4. GridPane	579
48.12.5. BorderPane	581
48.12.6. HBox i Vox	583
48.12.7. AnchorPane	586
48.13. Creació de nous controls	587
48.13.1. Crear un nou control per composició	588
48.13.2. Crear un nou control per herència	588
49. Inner classes i Nested classes	591
49.1. Què és una "nested class"	591
49.2. Representació d'una classe "nested" en UML	592
49.3. Per què necessitem les nested class?	592
49.4. Tipus de "inner classes"	593
49.4.1. "inner class" com a membre d'una altra classe	593

49.4.2. Classe "static nested class"	594
49.4.3. "inner class" local	595
49.4.4. "inner class" anònima	596
49.5. Exemple 1 - Inner class	597
49.6. Exemple 2 - Local class	598
49.7. Exemple 3 - Classe anònima	599
50. Interfícies funcionals	601
50.1. Com encapsular una funció en una variable	601
50.2. Proposta solució - versió 1	601
50.3. Proposta solució - versió 2	602
50.4. Proposta solució - versió 3	603
50.5. Interfícies funcionals	603
50.6. Mètodes default a les interfícies funcionals	604
50.7. Mètodes static a les interfícies funcionals	605
51. Expressions Lambda	607
51.1. Estructura de la llista d'arguments	608
51.2. Estructura del cos de la expressió lambda	609
51.3. Abast d'una expressió Lambda	610
51.4. Predicats	611
51.5. Exemples	612
51.5.1. Exemple 1	612
51.5.2. Exemple 2	612
51.5.3. Exemple 3	612
51.5.4. Exemple 4	613
51.5.5. Exemple 5	613
52. Exemple - Classes anònimes i Lambdes	615
52.1. Aproximació 1: Crear mètodes que busquin membres per una característica	618
52.2. Aproximació 2: Crear mètodes de cerca més generals	619
52.3. Aproximació 3: Especificar el codi del criteri de cerca en una classe local	619
52.4. Aproximació 4: Especificar el codi del criteri de cerca en una interfície	620
52.5. Aproximació 5: Especificar el codi del criteri de cerca en una classe anònima	621

52.6. Aproximació 6: Especificar el codi del criteri de cerca en una expressió Lambda	621
52.7. Aproximació 7: Utilitzar interfícies funcionals estàndards i expressions Lambda	621
52.8. Aproximació 8: Utilitzar expressions Lambda a tota l'aplicació	622
52.9. Aproximació 9: Introduir genèrics	623
53. Fluxos	625
53.1. Iteracions vs Streams	626
53.1.1. Exemple - Cerca de productes versió Col·leccions	626
53.1.2. Exemple - Cerca de productes versió Streams	629
53.2. Creació de fluxos	630
53.2.1. Directament a partir dels seus elements	630
53.2.2. A partir d'una matriu	630
53.2.3. A partir d'una List	631
53.2.4. A partir d'una funció generadora	631
53.3. Convertir un Stream a una col·lecció	632
53.3.1. Recuperar els elements en una List	632
53.3.2. Recuperar els elements en una matriu	633
53.4. Operacions intermèdies	633
53.4.1. Stream.filter()	633
53.4.2. Stream.map()	634
53.4.3. Stream.sorted()	635
53.5. Operacions terminals	635
53.5.1. Stream.forEach()	636
53.5.2. Stream.collect()	636
53.5.3. Stream.match()	637
53.5.4. Stream.count()	638
53.5.5. Stream.reduce()	638
53.6. Operacions interruptibles	639
53.6.1. Stream.anyMatch ()	639
53.6.2. Stream.findFirst()	640
53.7. Streams paral·lels	641
54. JavaFX - Esdeveniments	643
54.1. Esdeveniments	643
54.1.1. Creant una classe que implementa EventHandler<ActionEvent>	645

54.1.2. Utilitzant una classe anònima	646
54.1.3. Utilitzant una expressió lambda	647
54.1.4. Centralitzant la gestió d'esdeveniments en un sol mètode	648
54.2. Alguns esdeveniments comuns	649
54.2.1. Esdeveniments del ratolí	649
54.2.2. Esdeveniments del teclat	650
55. Patró Model-vista-Controlador	653
55.1. Avantatges i problemes del model	654
55.1.1. Avantatges	654
55.1.2. Problemes	655
55.2. Esquelet del patró	655
55.2.1. El model	655
55.2.2. La vista	656
55.2.3. El controlador	656
55.2.4. El mètode main	657
55.3. Un exemple d'implementació	658
55.3.1. El model	658
55.3.2. La vista	659
55.3.3. El Controlador	662
55.3.4. El mètode main	663
56. Patró Observer	665
56.1. Definició del patró	665
56.2. Implementació en Java	665
57. Vincular atributs de dues classes	669
57.1. El codi complet	675
58. JavaFX - Binding	679
58.1. Les propietats de JavaFX	679
58.1.1. Vincular propietats	679
58.1.2. Implementació 1 - Utilitzant directament el patró observer	680
58.1.3. Implementació 2 - Utilitzant bindings	681
58.1.4. Explicació de l'exemple	683
58.1.5. Escoltar propietats	685
58.2. Interfície ObservableList	686
58.3. ObservableList i TableView	689

58.4. Exemple - TableView	690
59. JavaFX - FXML	697
59.1. Creació d'una vista	697
59.2. Carregar un fitxer FXML	698
59.3. Afegir un controlador	699
59.3.1. Crear la classe controlador	699
59.3.2. Posar nom als nodes de la vista	700
59.3.3. Crear variables al controlador pels nodes de la vista	701
59.3.4. Gestió d'esdeveniments	702
60. JavaFX FXML - Un exemple guiat	705
60.1. Implementació de la Vista	706
60.1.1. Paquets	706
60.1.2. Formulari ContactesMestre	706
60.1.3. Scene Builder	708
60.1.4. Disseny del formulari ContactesMestre	709
60.1.5. Creació del formulari principal	715
60.1.6. Implementació de la funció main	716
60.2. Implementació dels Model / Controlador	718
60.2.1. Model	718
60.2.2. L'origen de dades	720
60.2.3. ContacteController	722
60.2.4. Enllaçar l'aplicació principal amb ContactesMestreController	724
60.2.5. Vincular la vista al controlador	726
60.3. Esdeveniments	728
60.3.1. Resposta a canvis en la selecció de la Taula	728
60.3.2. Detecta canvis de selecció en la taula	729
60.3.3. El botó d'esborrar	730
60.3.4. Gestió d'errors	731
60.3.5. Diàlegs per a crear i editar contactes	732
60.3.6. Enllaçar la vista i el controlador	734
60.3.7. Obrint la finestra de diàleg	735
61. Entrada / Sortida	737
61.1. Paquets java.io i java.nio	737
61.2. Fluxos basats en bytes i fluxos basats en text	738
62. Lectura i escriptura de fitxers	741

62.1. Classes de la biblioteca de Java	742
62.2. Per què calen dos conjunts de classes?	742
62.3. Tractament seqüencial o aleatori	743
62.4. Java IO: Streams	743
62.5. Java IO: Readers i Writers	745
63. Treballar amb fitxers	749
63.1. Conèixer el directori de treball	749
63.2. "Paths" o rutes	750
63.3. La classe File	751
63.4. Creació d'un objecte File	755
63.5. Comprovar si el fitxer existeix	756
63.6. Comprovar si el fitxer és un directori	756
63.7. Ruta absoluta i ruta canònica	756
63.8. Crear, renombrar i eliminar fitxers	759
63.9. Treballar amb els atributs dels fitxers	760
63.10. Copiar un fitxer	761
63.11. Saber la mida d'un fitxer	761
63.12. Mostrar tots els directoris arrel	762
63.13. Mostrar tots els fitxers i directoris d'un directori	762
64. Lectura i escriptura de streams binaris d'accés seqüencial	765
64.1. Classe java.io.InputStream	765
64.1.1. Mètode read()	766
64.1.2. Final del stream	766
64.1.3. Mètode read(byte[])	766
64.1.4. Mètodes mark() i reset()	767
64.1.5. Exemple 1	768
64.1.6. Exemple 2	768
64.2. Exemple 3	769
64.3. Classe Java.io.OutputStream	770
64.3.1. Mètode write(int)	770
64.3.2. Mètode flush()	771
64.3.3. Mètode close()	771
64.4. Classe java.io.FileInputStream i FileOutputStream	771
64.5. Mètodes de la classe InputStream	773
64.6. Mètodes de la classe OutputStream	774
64.6.1. Exemple Xifrat del cèsar	775

65. Lectura i escriptura de fitxers de text a baix nivell	777
65.1. Tractament de possibles errors d'accés al fitxer	777
65.2. Classe java.io.Reader	778
65.2.1. Caràcters i Unicode	778
65.2.2. Llegir caràcters amb un Reader	779
65.3. Classe java.io.Writer	779
65.3.1. Caràcters i Unicode	780
65.4. Classe java.io.InputStreamReader	780
65.4.1. Mètode read()	782
65.4.2. Final de fitxer	782
65.4.3. Tancar un InputStreamReader	783
65.5. Classe java.io.OutputStreamWriter	783
65.5.1. Determinar la codificació de caràcters	784
65.5.2. Tancar un OutputStreamWriter	785
65.6. Classe java.io.FileReader	785
65.7. Classe java.io.FileWriter	786
65.7.1. Sobreescriure el fitxer o afegir dades al final	787
65.8. La classe java.io.BufferedReader	787
66. Lectura i escriptura de fitxers de text a alt nivell	789
66.1. Classe Scanner	789
66.2. Exemple lectura d'un fitxer de text mitjançant la classe Scanner ..	790
66.3. Mètodes de la classe Scanner	792
66.4. Classe java.io.PrintWriter	793
66.5. Utilització d'un objecte PrintWriter	793
66.6. Mètodes de la classe PrintWriter	795
66.7. Exemple d'utilització de PrintWriter	796
66.8. Llegir un fitxer de text per paraules	797
66.9. Classe Scanner i la codificació de caràcters	800
66.10. Llegir un fitxer de text caràcter a caràcter	800
66.11. Classificar caràcters	800
66.12. Llegir un fitxer de text línia a línia	800
66.13. Escanejar una cadena	800
66.14. Com obrir un PrintWriter per afegir dades	801
67. Lectura i escriptura en de fitxers binaris d'accés aleatori	803
67.1. Escriptura seqüencial sobre un fitxer aleatori	804
67.2. Classe RandomAccessFile	805

67.3. Mètodes de la classe RandomAccessFile	811
68. Expressions regulars	815
68.1. Sintaxi de les expressions regulars	815
68.2. Algunes expresions regulars d'exemple	817
68.3. Treballar amb expressions regulars	818
68.3.1. Aplicar expressions regulars sobre objectes String	818
68.3.2. Altres mètodes regex de la classe String	820
68.4. Classes específiques per les expressions regulars	820
68.4.1. La classe Pattern	821
68.4.2. La classe Matcher	821
68.5. Alguns exemples	822
68.5.1. Exemple 1	822
68.5.2. Exemple 2	823
68.5.3. Exemple 3	823
68.5.4. Exemple 4	824
Bibliografia	827

40

Resultats d'aprenentatge i criteris d'avaluació

1. Escriu programes que manipulin informació seleccionant i utilitzant els tipus avançats de dades facilitats pel llenguatge.
 1. Escriu programes que utilitzin taules (arrays).
 2. Reconeix les llibreries de classes relacionades amb la representació i manipulació de col·leccions.
 3. Utilitza les classes bàsiques (vectors, llistes, piles, cues, taules de Hash) per emmagatzemar i processar informació.
 4. Utilitza iteradors per recórrer els elements de les col·leccions.
 5. Reconeix les característiques i els avantatges de cada una de les col·leccions de dades disponibles.
 6. Crea classes i mètodes genèrics.
 7. Utilitza expressions regulars en la cerca de patrons en cadenes de text.
 8. Identifica les classes relacionades amb el tractament de documents XML.
 9. Dissenya programes que realitzen manipulacions sobre documents XML.
2. Gestiona els errors que poden aparèixer en els programes, utilitzant el control d'excepcions facilitat pel llenguatge.
 1. Reconeix els mecanismes de control d'excepcions facilitats pel llenguatge.

-
2. Implementa la gestió d'excepcions en l'ús de classes facilitades pel llenguatge.
 3. Implementa el llançament d'excepcions en les classes que desenvolupa.
 4. Reconeix la incidència de l'herència en la gestió d'excepcions.
3. Desenvolupa interfícies gràfiques d'usuari simples, utilitzant les llibreries de classes adequades.
 1. Utilitza les eines de l'entorn de desenvolupament per crear interfícies gràfiques d'usuari simples.
 2. Programa controladors d'esdeveniments.
 3. Escriu programes que utilitzin interfícies gràfiques per a l'entrada i sortida d'informació.
 4. Realitza operacions bàsiques d'entrada/sortida d'informació, sobre consola i fitxers, utilitzant les llibreries de classes adequades.
 1. Utilitza la consola per realitzar operacions d'entrada i de sortida d'informació.
 2. Aplica formats en la visualització de la informació.
 3. Reconeix les possibilitats d'entrada/sortida del llenguatge i les llibreries associades.
 4. Utilitza fitxers per emmagatzemar i recuperar informació.
 5. Crea programes que utilitzen diversos mètodes d'accés al contingut dels fitxers.

41

Continguts

1. Aplicació de les estructures d'emmagatzematge en la programació orientada a objectes:
 1. Estructures de dades avançades.
 2. Creació d'arrays.
 3. Arrays multidimensionals.
 4. Cadenes de caràcters.
 5. Col·leccions i iteradors.
 6. Classes i mètodes genèrics.
 7. Manipulació de documents XML. Expressions regulars de cerca.
2. Control d'excepcions:
 1. Cadenes de caràcters.
 2. Captura d'excepcions.
 3. Captura enfront de delegació.
 4. Llançament d'excepcions.
 5. Excepcions i herència.
3. Interfícies gràfiques d'usuari:
 1. Creació i ús d'interfícies gràfiques d'usuari simples.
 2. Concepte d'esdeveniment. Creació de controladors d'esdeveniments.
 3. Paquets de classes per al disseny d'interfícies.

4. Lectura i escriptura d'informació:

1. Tipus de fluxos. Fluxos de bytes i de caràcters.
2. Classes relatives a fluxos. Utilització de fluxos.
3. Entrada/sortida. Llibreries associades.
4. Fitxers de dades. Registres.
5. Gestió de fitxers.
 1. Modes d'accés.
 2. Lectura/escriptura.
 3. Utilització dels sistemes de fitxers.
 4. Creació i eliminació de fitxers i directoris.

Control d'excepcions

Hi ha dos aspectes importants quan es tracta amb errors de programa, per una banda la **detecció** i per l'altra la **gestió**.

El control d'excepcions és un mecanisme que permet que un programa gestioni situacions excepcionals i continuï la seva execució normal.

Els **Errors d'execució** succeeixen mentre un programa s'està executant i detecta una operació que és impossible de tirar endavant.

Per exemple, si s'accedeix a un index fora de rang d'una matriu es produirà un error d'execució de tipus **ArrayIndexOutOfBoundsException**, si s'introduceix un valor **double** quan s'espera un valor **int** es produirà un error d'execució de tipus **InputMismatchException**.

Els errors d'execució en Java es representen mitjançant uns objectes anomenats **excepcions**.

42.1. Llançament d'excepcions

Quan es detecta una condició d'error que no pot gestionar adequadament el codi que l'ha detectat, el que cal fer, és avisar d'aquesta condició amb l'esperança que un codi de nivell superior, que si és capaç de gestionar l'error, rebi la notificació i actui en conseqüència.

En aquest cas, en la terminologia Java, el que es faria és **llançar una excepció** adequada amb la informació necessària per a poder gestionar l'error.

Per exemple, suposem que algú intenta retirar massa diners d'un compte corrent:

```
if (quantitat > balance) {  
    // I ara què?  
}
```

En primer lloc es cerca una classe **Exception** adequada, la llibreria de Java en proporciona un munt que permeten notificar tot tipus de condicions excepcionals organitzades en un esquema jeràrquic, i a continuació es **llença** l'excepció amb la clàusula **throw**.

```
if (quantitat > balance) {  
    throw new IllegalArgumentException("La quantitat és major que el  
    balanç");  
}
```

Quan es llença una excepció, **la execució no continua amb la següent instrucció**, el flux normal d'execució s'interromp.

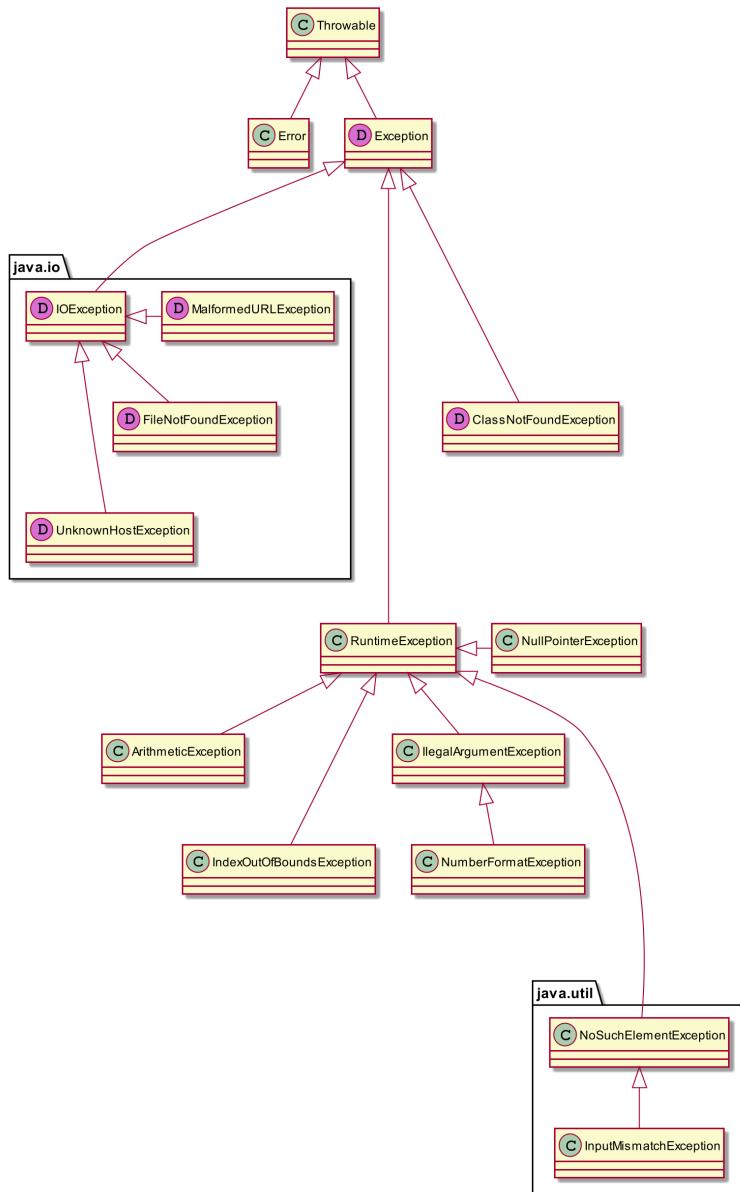


Figura 42.1. Part de la jerarquia de les classes d'excepcions

42.2. Excepcions

A Java, les excepcions que es poden llançar i capturar es poden classificar en tres categories.

Errors interns

Representats per tots els descendents de la classe **Error**. Si es dona algun d'aquests errors, cosa poc habitual, poca cosa s'hi pot fer a part de notificar a l'usuari i sortir de l'aplicació.

Per exemple, en aquesta categoria s'hi troben els errors de falta de memòria, **OutOfMemoryError** i els errors de la màquina virtual, **VirtualMachineError**.

Excepcions descendents de RuntimeException

Representen **errors de programació** com ara accedir a una posició incorrecte d'una matriu, **IndexOutOfBoundsException**, Dividir un enter per zero, **ArithmaticException** o passar un paràmetre incorrecte a un mètode, **IllegalArgumentException**.

Tots els altres descendents de la classe Exception

Aquestes excepcions s'anomenen **checked exceptions**. Indiquen que alguna cosa ha anat malament **per alguna raó fora del control del programador**.

Per exemple, no s'ha trobat el fitxer indicat, **FileNotFoundException** o s'ha perdut la connexió de xarxa, **ConnectException**.

42.3. Com es gestionen les "checked exception"

Java **es pren molt seriosament les "checked exceptions"**, tan és així que **obliga** al programador a gestionar-les. És a dir, **es produeix un error de compilació si es deixa de gestionar alguna "checked exception"**.

Per exemple, el següent fragment codi no compila:

```
public static String readData(String filename)
    File inFile = new File(filename);
    Scanner in = new Scanner(inFile); ①
    . .
}
```

- ① Aquesta línia pot llençar una excepció **FileNotFoundException**, que com que és una "checked exception" **s'ha de gestionar obligatoriament**.

Per resoldre el problema anterior tenim dues opcions:

1. **Podem** gestionar la excepció i per tant la capturem i la tractem.

```
public static String readData(String filename)
try {
    File inFile = new File(filename);
    Scanner in = new Scanner(inFile);
    . .
} catch (FileNotFoundException ex) {
    . .
}
```

2. **No sabem** com gestionar l'excepció i per tant la deixem passar amb l'esperança que algun dels mètodes de la pila de crides a funcions pugui fer-ho.

```
public static String readData(String filename) throws
FileNotFoundException
File inFile = new File(filename);
Scanner in = new Scanner(inFile);
. .
}
```



En aquest cas el mètode que ha cridat a **readData** estaria obligat a gestionar o deixar passar l'excepció explícitament.

42.4. Com es gestionen les RuntimeException

Les excepcions que deriven de **RuntimeException** representen errors de programació per tant **la millor manera de tractar-les és modificar el codi perquè no es produueixin**.

Aquestes excepcions és sempre possible evitar-les si es comprova de manera exhaustiva les condicions que porten a llançar-les.

No obstant si que és útil utilitzar una estructura try-catch com a xarxa de seguretat per si, com a programadors, ens hem descuidat de comprovar alguna de les condicions que produeixen l'excepció.

Seguim tenint dues opcions alhora de tractar aquestes excepcions:

1. **Podem** gestionar la excepció i per tant la capturem i la tractem.

```
public static String add(String valor, int index)
    try {
        this.valors[index] = valor;
        .
        .
    } catch (IndexOutOfBoundsException ex) { ①
        .
    }
}
```

- 1 S'accedeix a una posició de la matriu fora dels seus rangs.
- 2. **No sabem** o a diferència d'elles "checked exceptions", **no volem** gestionar l'excepció i per tant la deixem passar amb l'esperança que algun dels mètodes de la pila de crides a funcions pugui fer-ho.
En el cas de les **RuntimeExceptions** no és necessària la clàusula **throws** a la declaració del mètode que dispara l'excepció.

```
public static String add(String valor, int index)
    this.valors[index] = valor; ①
}
```

- 1 Si l'index proporcionat queda fora del rang de la matriu és llençarà una excepció **IndexOutOfBoundsException** automàticament.

42.5. Captura d'excepcions

La construcció **try/catch** del llenguatge Java permet capturar una excepció llençada.

Una excepció no capturada es propaga stack avall fins arribar al final on es manifesta amb la interrupció del programa.

El bloc try/catch funciona de la següent manera:

```
try {
    // Bloc de codi que pot generar una excepció
} catch (Excepció que es vol capturar) {
    // Accions que es prenen en cas que la excepció capturada
```

```
// coincideix amb la que s'ha disparat.
} catch (Excepció que es vol capturar) {
    // Accions que es prenen en cas que la excepció capturada
    // coincideix amb la que s'ha disparat.
} catch (...) {
    ...
}
```

Per exemple,

```
try {
    String filename = . . .;
    Scanner in = new Scanner(new File(filename)); ①
    String input = in.next(); ②
    int value = Integer.parseInt(input); ③
    ...
} catch (IOException exception) {
    exception.printStackTrace();
} catch (NumberFormatException exception) {
    System.err.println(exception.getMessage());
}
```

El fragment de codi anterior pot llançar tres excepcions:

- ① L'objecte **Scanner** pot llançar una excepció **FileNotFoundException**.
 - ② El mètode **next()** de l'objecte **scanner** pot llançar una excepció **NoSuchElementException**.
 - ③ **Integer.parseInt()** pot llançar una excepció de tipus **NumberFormatException**.
- Si **qualsevol de les excepcions** anterior és llançada, aleshores es saltarien la resta d'instruccions del block **try**.
 - Si **no es llança** cap excepció, el bloc de codi dins dels **catch** s'ignora.
 - Si l'excepció llançada fos **FileNotFoundException** s'activaria el **catch(IOException)** ja que si ens fixem en l'estructura jeràrquica de les Excepcions la primera excepció és un cas particular de la segona. De fet, qualsevol de les excepcions derivades de **IOException** serien capturades pel **catch** anterior.

- Si l'excepció llançada fos **NumberFormatException** seria la segona clàusula **catch** la que s'executaria.
- L'excepció **NoSuchElementException** no apareix a cap clàusula **catch** ni deriva de **IOException** ni de **NumberFormatException**, per tant, aquesta excepció si es produís no seria capturada i el flux d'execució pujaria un nivell a la pila de crides a funcions en espera de ser capturada allà.



Només s'haurien de capturar les excepcions que es puguin gestionar. En cas de no tenir les eines per a gestionar-les és preferible deixar-les passar.



Les diferents excepcions mantenen una jerarquia, capturar una excepció d'un nivell superior capture automàticament totes les excepcions que en deriven.

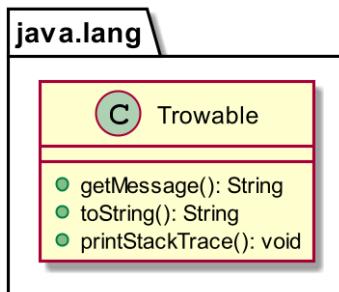


L'ordre de captura de les excepcions especificat en els blocs **catch** és important. És generarà un **error de compilació** si l'ordre de les clàusules **catch** no coincideix amb l'ordre jeràrquic de les excepcions.

42.6. Obtenir informació de les excepcions

Un objecte **Exception** conté informació important sobre l'excepció llançada.

Els següents mètodes es poden utilitzar per obtenir informació independentment de l'excepció.



getMessage(): String

Retorna el missatge que descriu la excepció capturada.

toString(): String

Retorna la concatenació de dues cadenes, el nom sencer de l'excepció i el valor del mètode `getMessage()`

printStackTrace(): void

Mostra l'objecte **Throwable** i el contingut de la pila de crides a funcions.

42.7. Clàusula finally

A vegades serà necessari fer alguna acció independentment de si s'ha produït o s'ha capturat una excepció.

La construcció **finally** s'utilitza per gestionar aquesta situació.

Per exemple, és important tancar un *PrintWriter* per assegurar que tota la informació és guardada al disc.

En el següent exemple es possible no arribar mai a la línia 3 i per tant no tancar mai l'objecte *PrintWriter*, això pot passar en cas que es produueixi una excepció a la línia 1 o 2.

```
PrintWriter out = new PrintWriter(filename);
writeData(out);
out.close();
```

Per solucionar-ho s'utilitza la clàusula *finally*,

En aquest exemple només volem assegurar el tancament del objecte PrintStream però no gestionem cap excepció:

```
PrintWriter out = new PrintWriter(filename);
try {
    writeData(out);
} finally {
    out.close();
}
```

En aquest exemple només volem assegurar el tancament del objecte PrintStream i a més capturem les possibles excepcions:

```
PrintWriter out = new PrintWriter(filename);
```

```
try {
    writeData(out);
} catch (IOException ex) {
    ...
} finally {
    out.close();
}
```

El codi que es troba dins del bloc **finally** s'executa **sempre** independentment de si es produeix una excepció o de si es captura una excepció.

- Si no es llença cap excepció dins del bloc **try** s'executa el bloc **finally** i l'execució segueix després del bloc **try**.
- Si es llença una excepció dins del bloc **try** que és capturada dins d'un bloc **catch**, s'executa el bloc **catch** en primer lloc, després el bloc **finally** i finalment l'execució del programa segueix després del bloc **try**.
- Si es llença una excepció dins del bloc **try** i no es capture a cap dels blocs **catch** s'executa el codi dins del bloc **finally** i l'excepció és passa a qui ha cridat el mètode on s'ha produït.



El bloc **finally** s'executa fins i tot si apareix un **return** abans d'arribar al bloc **finally**. Per exemple,

```
PrintWriter out = new PrintWriter(filename);
try {
    writeData(out);
    return; // No evita l'execució de finally!!
} catch (IOException ex) {
    ...
} finally {
    out.close();
}
```

42.8. Definir noves classes d'excepció

Java proporciona un conjunt de classes d'excepció, no obstant és possible que hi hagi situacions en que cap de les excepcions proporcionades per Java s'ajusti a les nostres necessitats, per solventar-ho **Java permet definir noves classes d'excepció**.

Per definir una nova excepció n'hi ha prou en programar una classe que derivi de la classe **Exception**, **RuntimeException** o qualsevol de les excepcions predefinides del llenguatge.

En general es deriva de les classes **Exception** o **RuntimeException** recordeu que la diferència principal és que derivar de la classe **Exception** provoca que la nova excepció sigui una **checked exception** i per tant **és obligatori tractar-la** i derivar de **RuntimeException** no.

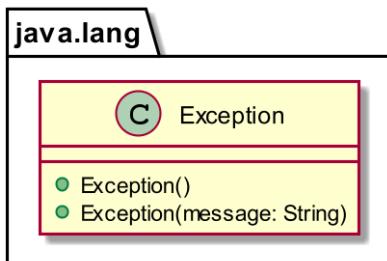


Les "checked exceptions", és a dir les excepcions que deriven directament de la classe **Exception** han generat controvèrsia a la comunitat de Java des del principi. De fet, hi ha programadors que no les utilitzen i les ignoren rellançant-les com a excepcions no "checked". Per exemple:

```
try {
    // bla bla bla
} catch (CheckedException e) {
    throw new RuntimeException(e);
}
```

Assegureu-vos que enteneu les implicacions de derivar de les classes **Exception** o **RuntimeException** abans de fer-ho.

A part d'això, la majoria d'excepcions tenen dos constructors i es recomana afegir-los per les noves excepcions, aquests són els següents:



Per exemple, el següent codi defineix una nova excepció:

```

class InvalidDniException extends RuntimeException {

    private String dni;

    public InvalidDniException(String dni) {
        super("DNI incorrecte: " + dni);
        this.dni = dni;
    }

    public String getDni() {
        return dni;
    }
}

```

42.9. Construcció try-amb-recursos

La construcció **try-amb-recursos**, **try-with-resources** en anglès, és un **try** que declara un o més recursos.

Un **recurs** és un objecte que **ha de ser tancat** un cop el programa deixa de necessitar-lo. Tècnicament és un objecte que implementa la interfície **java.lang.AutoCloseable**, que inclou a tots els objectes que implementen la interfície **java.io.Closeable**.

La construcció *try-amb-recursos* garanteix que tots els recursos es tanquen al final de la construcció.

La sintaxi és la següent:

```

try (declarar i crear els recursos) {
    utilitzar els recursos
}

```

Vegem un primer exemple amb un codi artificial:

```

public class TestTryWithResources {

    public static void main(String[] args) {

        System.out.println("Test 1"); ①
        try (TestAutoCloseable t = new TestAutoCloseable()) {

```

```

        t.testMethod();
    } catch (Exception ex) {

    }

    System.out.println("Test 2"); ②
    try (TestAutoCloseable t = new TestAutoCloseable()) {
        t.testMethod();
        throw new Exception();
    } catch (Exception ex) {

    }
}

class TestAutoCloseable implements AutoCloseable {

    @Override
    public void close() throws Exception {
        System.out.println("S'ha cridat a Close()");
    }

    public void testMethod() {
        System.out.println("testMethod()");
    }

}

```

- ① Comprovem que es crida el mètode **close()** en una execució sense errors del bloc **try-catch**.
- ② Comprovem que es crida el mètode **close()** en una execució amb errors del bloc **try-catch**.

```

Test 1
testMethod()
S'ha cridat a Close()
Test 2
testMethod()
S'ha cridat a Close()

```

I un segon exemple amb codi real:

```
static void gestionarFitxer(String nomFitxer) {
```

```
File file = new File(nomFitxer);
RandomAccessFile raf = null;
try {
    try {
        raf = new RandomAccessFile(file, "rw");
        // Fer coses
    } finally {
        raf.close();
    }
} catch (IOException ex) {
    ex.printStackTrace();
}
}
```

Quedaria de la següent manera:

```
static void gestionarFitxer(String nomFitxer) {
    File file = new File(nomFitxer);

    try (RandomAccessFile raf = new RandomAccessFile(file, "rw")) {
        /* Fer coses */
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}
```

42.10. Algunes recomanacions a l'hora de treballar amb excepcions

42.10.1. Llençar d'hora capturar tard

Quan un mètode detecta un problema que no pot resoldre és millor llançar una excepció que intentar solucionar el problema amb una solució imperfecte.

Per exemple, suposeu que un mètode intenta llegir un número d'un fitxer i el fitxer no conté cap número, retornar un zero en aquesta situació és una mala solució ja que s'està emmascarant un problema i segons com pot generar problemes en altres parts del codi.

De la mateixa manera, un mètode només hauria de capturar una excepció si realment pot solucionar la situació, si no, és millor propagar l'excepció a l'espera que aparegui algú més competent per gestionar-la.

42.10.2. No emmascarar excepcions

Quan un mètode llença una "checked exception" i no s'ha gestionat el compilador es queixa. Si no es té massa clar que fer amb l'excepció i es vol seguir treballant es pot tenir l'impuls de silenciar el compilador capturant l'excepció sense fer res més.

```
try {
    Scanner in = new Scanner(new File(filename));
    // El compilador es queixa de FileNotFoundException
    ...
} catch (FileNotFoundException e) {} // Aquí!!!
```

És **molt mala idea** fer això, les excepcions estan dissenyades per a transmetre informació d'un problema a un gestor competent, si les emmascaren es seguiran produint però ni es gestionaran ni ens donaran informació cosa que a la llarga és serà un problema.

42.10.3. Tenir en compte que algunes excepcions es poden produir al bloc finally

```
File file = new File("nomFitxer");
RandomAccessFile raf = null;
try {
    raf = new RandomAccessFile(file, "rw");
    // Fer coses
} catch (FileNotFoundException ex) {
    ex.printStackTrace();
} finally {
    raf.close();
}
```

El problema amb el codi anterior és que el mètode *close()* pot produir una excepció.

Tenir en compte que algunes excepcions
es poden produir al bloc **finally**

Per gestionar aquesta excepció es pot afegir un bloc *try/catch* encapsulant el mètode **close()**.

```
File file = new File("nomFitxer");
RandomAccessFile raf = null;
try {
    raf = new RandomAccessFile(file, "rw");
    // Fer coses
} catch (FileNotFoundException ex) {
    ex.printStackTrace();
} finally {
    try {
        raf.close();
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}
```

Però la millor manera millor manera és utilitzar un bloc **try/finally** per tancar els recursos i un bloc **try/catch** per gestionar els errors.

Per exemple,

```
File file = new File("nomFitxer");
RandomAccessFile raf = null;
try {
    try {
        raf = new RandomAccessFile(file, "rw");
        // Fer coses
    } finally {
        raf.close();
    }
} catch (IOException ex) {
    ex.printStackTrace();
}
```

O encara millor, utilitzar una construcció **try-amb-recursos**.

```
static void gestionarFitxer(String nomFitxer) {
    File file = new File(nomFitxer);

    try (RandomAccessFile raf = new RandomAccessFile(file, "rw")) {
```

Tenir en compte que algunes excepcions
es poden produir al bloc **finally**

```
/* Fer coses */  
} catch (IOException ex) {  
    ex.printStackTrace();  
}  
}
```

43

La classe ArrayList

Per emmagatzemar una col·lecció d'objectes podem utilitzar una **matriu** però un cop creada la matriu la seva mida és fixa.

Java proporciona la classe **ArrayList** que funciona essencialment com una matriu amb dues diferències:

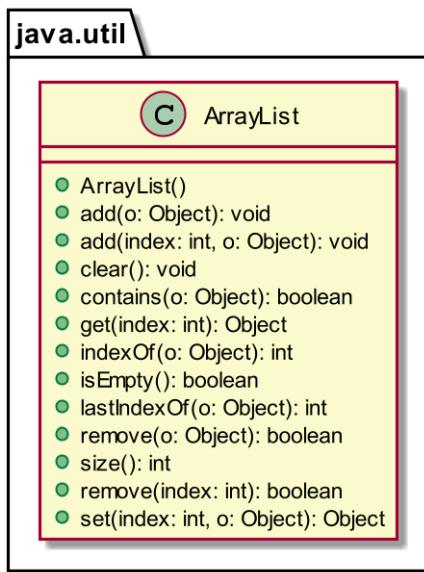
1. **permet canviar la seva longitud dinàmicament.**
2. Només admet objectes, és a dir no admet **tipus primitius** cosa que no és massa problemàtica ja que tenim classes equivalents per a cada tipus primitiu.

Existeixen dues implementacions diferents de la classe **ArrayList**, una és la que es trobava a les biblioteques de Java abans de la versió Java 5 i l'altra és la versió actual que apareix a partir de Java 5.

A nivell de funcionalitat les dues implementacions són pràcticament iguals, la diferència fonamental la comentarem una mica més endavant.

43.1. `java.util.ArrayList` versió anterior a Java 5

El següent diagrama UML mostra els mètodes més comuns de la classe **ArrayList** la **signatura** dels mètodes és diferent en les diferents implementacions però la funcionalitat és la mateixa, vegem-ho:



ArrayList()

Crea una llista buida.

add(o: Object): void

Afegeix un **Object** al final de la llista.

add(index: int, o: Object): void

Afegeix un **Object** a la posició **index** de la llista.

clear(): void

Elimina tots els elements de la llista.

contains(o: Object): boolean

Retorna **true** si la llista conté l'objecte **o**.

get(index: int): Object

Retorna l'objecte amb index **index** de la llista

indexOf(o: Object): int

Retorna la posició, l'**index**, on es troba la primera aparició de l'objecte **o** dins de la llista.

isEmpty(): boolean

Retorna **true** si la llista no té elements.

lastIndexOf(o: Object): int

Retorna la posició, l'index, on es troba la primera aparició de l'objecte **o** dins de la llista **però començant pel final de la llista**.

remove(o: Object): boolean

Elimina l'objecte **o** de la llista.

size(): int

Retorna el nombre d'elements de la llista.

remove(index: int): boolean

Elimina l'objecte amb index **i** de la llista.

set(index: int, o: Object): Object

Mou l'objecte **o** a la posició **index** dins de la llista.

El **punt important** en aquesta implementació d'**ArrayList** és que els elements que s'emmagatzemem a la llista es **tracten com a Object** i per tant **en una sola llista podem posar-hi qualsevol tipus d'objecte**. Per exemple, el següent codi és perfectament vàlid:

```
import java.util.ArrayList;

public class TestArrayList {

    public static void main(String[] args) {
        ArrayList m = new ArrayList();

        Integer i1 = new Integer(10);
        m.add(1);
        int i2 = 20;
        m.add(i2); // converteix automàticament i2 en un Integer
        (autoboxing)
        String s1 = "cadena";
        m.add(s1);

        for (int i = 0; i < m.size(); i++) {
            System.out.println(m.get(i).toString());
        }
    }
}
```

```
1
20
cadena
```

Els problemes amb la implementació anterior és bàsicament un, **quan intentem recuperar un valor hem de saber de quin tipus és** i fer un cast ja que si no, no podrem accedir als seus mètodes, per exemple:

```
import java.util.ArrayList;

public class TestArrayList {

    public static void main(String[] args) {
        ArrayList m = new ArrayList();

        String s1 = "cadena";
        m.add(s1);

        String valor = m.get(0); ①
    }
}
```

- ① L'última línia dóna error de compilació. Encara que l'objecte que estem referenciant és realment un **String**, si s'ha guardat com a tipus **Object**, el compilador no té forma de saber que realment és un **String**.

Fixeu-vos que assignar un objecte de tipus **String** a una referència de tipus **Object** és correcte, perquè **String** deriva d'**Object**, al revés però no és sempre cert.

Però nosaltres sí que estem segurs que l'objecte en qüestió és un **String!** Aleshores, li podem dir al compilador que ja sabem què estem fent, i que deixi de queixar-se. Això es fa amb l'anomenat **cast**:

```
String valor = (String) m.get(3);
```

En el nostre petit programa això és suficient, però què passa si en un programa més gran i complex hem afegit una referència a la llista que no és de tipus **String**?

```
ArrayList m = new ArrayList();
```

```

String s = "cadena";
m.add(s);
Integer i = new Integer(10);
m.add(i);

String valor1 = (String) m.get(0);
String valor2 = (String) m.get(1); ①

```

- ① El compilador no es queixa, perquè li estem dient que confiï en nosaltres i que converteix la referència d'**Object** a **String**. Però quan executem:

```

Exception in thread "main" java.lang.ClassCastException:
java.lang.Integer cannot be cast to java.lang.String
at TestArrayList.main(TestArrayList.java:15)

```

El programa “peta” en temps d’execució, amb una excepció de tipus **ClassCastException**, hem intentat convertir quelcom que realment era un **Integer** a un **String**.

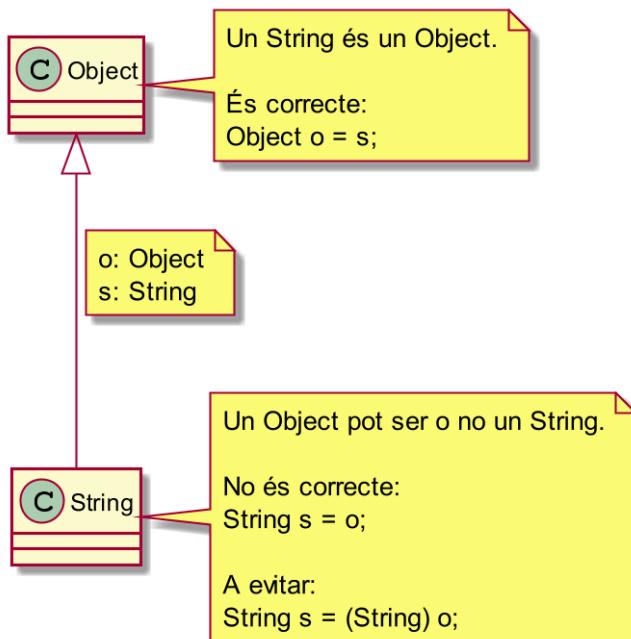


Figura 43.1. Relació entre String i Object

Tenim diverses maneres de gestionar aquesta possibilitat. O abans de fer la conversió ens assegurem que aquesta és correcte:

```
ArrayList m = new ArrayList();

String s = "cadena";
m.add(s);
Integer i = new Integer(10);
m.add(i);

String valor1 = (String) m.get(0);
if (m.get(1) instanceof String) {
    String valor2 = (String) m.get(1);
} else {
    // Sabem que m.get(1) no és una cadena i fem el que toqui ...
}
```

O bé capturem l'excepció i actuem en conseqüència:

```
ArrayList m = new ArrayList();

String s = "cadena";
m.add(s);
Integer i = new Integer(10);
m.add(i);

String valor1 = (String) m.get(0);
try {
    String valor2 = (String) m.get(1);
} catch (ClassCastException e) {
    // Codi per gestionar l'error
}
```

Amb això resolem les possibles petades del programa, però no és gratuït:

- Per una banda, seguim tenint el problema que en algun punt del programa es pugui introduir a la llista un objecte de tipus incorrecte, i aquest error només es detecta quan l'objecte s'extreu de la llista, de manera que ens pot costar bastant trobar l'origen de l'error.

- Per altra banda, el codi, que era molt senzill a la primera versió de l'exemple, s'allarga i es complica amb el tractament de les situacions anòmals, i es fa menys llegible.

De tot això n'hem de treure dues importants lliçons:

- **Els elements que introduïm en una llista han de ser del tipus més restringit possible:** és una mala idea utilitzar una mateixa llista per guardar objectes de tipus molt diferent.
- **Cal reservar els cast per aquelles poques situacions en què són imprescindibles**, i evitar-los en la resta de casos.

Per aquesta raó a partir de Java 5 la implementació de les classes col·lecció canvien amb l'aparició d'unes noves construccions anomenades **genèrics**.

43.2. java.util.ArrayList versió posterior a Java 5

El problema introduït a l'apartat anterior va motivar que en la versió 5 de Java s'introduís la noció dels genèrics.

Una possible solució al problema del tipus seria crear classes **ArrayList** diferents per a cada tipus de dada. Per exemple, si volem una **ArrayList** d'enters, tindríem la classe **IntegerArrayList**, que només podria rebre elements de tipus **Integer** i retornaria elements de tipus **Integer**, la classe **StringArrayList**, per treballar amb llistes de **String**, etc...

Això resoldria el nostre problema, però caldria reimplementar la llista per a cada tipus de dada que volguéssim guardar. Acabaríem tenint un munt de classes diferents que fan pràcticament el mateix, però canviant el tipus de dades amb què treballen.

No seria genial poder passar aquest tipus de dada com a paràmetre quan creem l'objecte llista, i que la mateixa implementació ens servís per a crear llistes per a qualsevol d'ells? Doncs aquesta és justament la idea dels genèrics.

Naturalment, passar un tipus com a paràmetre no és exactament el mateix que passar un valor, i la sintaxi canvia una mica. A partir d'ara, en totes les referències que utilitzem a tipus de **ArrayList**, i en tots els **new**, especificarem entre **\<\>** el tipus de dada adequat:

```
ArrayList<String> m = new ArrayList<String>();  
  
String s = "cadena";  
m.add(s);  
  
String valor1 = m.get(0); // No necessitem casting
```

En aquest exemple s'ha eliminat la necessitat del cast, i ara sentències com:

```
Integer i = 3;  
m.add(i); // ERROR
```

O

```
Integer valor1 = m.get(0); // ERROR
```

donen error de compilació. Si intentem introduir o extreure una referència de la llista de tipus invàlid, ho notarem al mateix moment d'escriure el codi, i podrem arreglar l'error immediatament. Ja no caldrà comprovar els tipus ni gestionar aquestes excepcions.

44

Genèrics

Els **genèrics** permeten que els **tipus** (classes i interfícies) siguin paràmetres quan es defineixen classes, interfícies o mètodes.

De la mateixa manera que els paràmetres dels mètodes proporcionen una manera de reutilitzar el codi amb diferents entrades, els paràmetres "tipus" permeten reutilitzar les declaracions de les classes per obtenir classes amb diferents tipus de dades.

Amb l'ajuda dels genèrics és possible crear classes, interfícies i mètodes que treballin amb seguretat de tipus amb diferents tipus de dades. Són molt útils si tenim en compte que molts algoritmes són idèntics lògicament independentment del tipus e dades sobre els que s'apliquen, en aquesta situació, els genèrics permeten **definir l'algoritme una vegada i aplicar-lo sobre diferents tipus de dades sense esforços addicionals**.



Java sempre ha proporcionat una manera de crear classes, interfícies i mètodes "generalitzats" **utilitzant referències de tipus Object**.

El problema d'aquesta manera de procedir és la impossibilitat de treballar amb **seguretat de tipus**.

Utilitzar genèrics proporciona els següents avantatges:

- Permeten transformar errors en temps d'execució en errors en temps de compilació facilitant la depuració dels programes.

Imaginem, per exemple, que volem una llista d'objectes **Comanda** i inadvertidament afegim un objecte de tipus diferent dins la llista, una cosa similar a la següent:

```
List comandes = new ArrayList();

comandes.add(new Comanda());
comandes.add(new Comanda());
comandes.add(new Integer());
comandes.add(new Comanda());

for (Comanda c in comandes) {
    System.out.println(c.getNumComanda()); ①
}
```

- ① L'exemple anterior **compila correctament** però genera **un error en temps d'execució** a l'intentar accedir al mètode **getNumComanda()** del tercer element.

Utilitzant genèrics tindríem:

```
List<Comanda> comandes = new ArrayList<Comanda>();

comandes.add(new Comanda());
comandes.add(new Comanda());
comandes.add(new Integer()); ①
comandes.add(new Comanda());

for (Comanda c in comandes) {
    System.out.println(c.getNumComanda());
}
```

- ① Automàticament el compilador detecta un error en aquest punt ja que la col·lecció només admet objectes de tipus **Comanda**.
- Desapareix la necessitat d'utilitzar casts. Per exemple, el següent codi:

```
List list = new ArrayList();
list.add("holà");

String s = (String) list.get(0); ①
```

- ❶ Aquest cast és obligatori.

Utilitzant genèrics estalviem el cast:

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);
```

44.1. Tipus genèrics

Un *tipus genèric* és una classe o una interfície parametrizada amb tipus.

44.2. Implementació de la classe Box

Per exemple, volem implementar una classe **Box** que pugui encapsular objectes de qualsevol tipus.

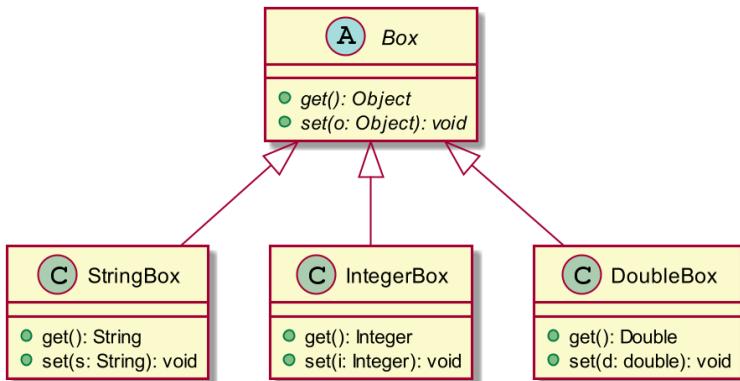
```
public class Box {
    private Object object;

    public void set(Object object) {
        this.object = object;
    }

    public Object get() {
        return object;
    }
}
```

El **problema** és que amb l'enfoc anterior no tenim cap control del contingut dels objectes **Box**, si una part del codi espera un objecte **Box** amb un **String** a l'interior però l'objecte proporcionat conté un **Integer** amb sort es genererà un error d'execució.

Una manera de solventar-ho podria ser el següent plantejament, crear una classe concreta per a cada possible contingut de **Box**:



El problema principal d'aquesta opció és la poca mantenibilitat de tot plegat, no obstant els genèrics ens proposen una solució més senzilla.

44.3. La classe Box utilitzant genèrics

Una *classe genèrica* es defineix amb el següent format:

```
class nom<T1, T2, ..., Tn> { /* ... */ }
```

La secció delimitada per <> que segueix al nom de la classe especifica el tipus dels paràmetres de la classe.

Per exemple, utilitzant la construcció anterior la classe **Box** queda:

```
public class Box<T> {
    private T t;

    public void set(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }
}
```

Com es pot veure, totes les ocurrències de **Object** s'han reemplaçat per **T**.



Una *variable tipus* pot ser de qualsevol tipus **no primitiu**: una classe, una interfície, una matriu o fins i tot una altra *variable tipus*.

44.4. Noms de les *variables tipus*

Per convenció, el nom dels *paràmetres tipus*, està format per **una única lletra en majúscula**. Els noms més comuns dels paràmetres són:

- E - Element
- K - Key
- N - Number
- T - Type
- V - Value
- S,U,V etc. - segon, tercer i quart tipus.

44.5. Referències a classes genèriques

Podem fer referència a la classe **Box** des del nostre codi caldrà substituir els **paràmetres tipus** per **arguments tipus**, això és, substituir els *paràmetres tipus* per tipus concrets.

Per exemple:

```
Box<Integer> integerBox;
integerBox = new Box<Integer>();
```

A partir de Java 7 també es pot utilitzar la següent sintaxi:

```
Box<Integer> integerBox;
integerBox = new Box<>(); ①
```

- ① Aquesta construcció és correcte sempre i quant el compilador pugui inferir els *arguments tipus* del context.

44.6. Classes genèriques amb més d'un paràmetre

Una classe genèrica pot tenir més d'un paràmetre, per exemple:

Utilitzar classes genèriques
sense arguments tipus

```
public interface Parell<K, V> {  
    public K getKey();  
    public V getValue();  
}  
  
public class ParellOrdenat<K, V> implements Parell<K, V> {  
  
    private K key;  
    private V value;  
  
    public ParellOrdenat(K key, V value) {  
        this.key = key;  
        this.value = value;  
    }  
  
    public K getKey() {  
        return key;  
    }  
  
    public V getValue() {  
        return value;  
    }  
}
```

Podem crear instàncies de la classe anterior:

```
Parell<String, Integer> p1 = new ParellOrdenat<String,  
Integer>("edat", 3);  
Parell<String, String> p2 = new ParellOrdenat<String,  
String>("nom", "Pere");
```

O utilitzant l'operador diamant:

```
Parell<String, Integer> p1 = new ParellOrdenat<>("edat", 3);  
Parell<String, String> p2 = new ParellOrdenat<>("nom", "Pere");
```

44.6.1. Utilitzar classes genèriques sense arguments tipus

Donada la següent classe:

```
public class Box<T> {
```

```

private T t;

public void set(T t) {
    this.t = t;
}

public T get() {
    return t;
}
}

```

Java permet instanciar una classe genèrica, com la anterior, sense especificar els *arguments tipus*, per exemple, les següents construccions són correctes:

```
Box rawBox = new Box();
```

```
Box<String> stringBox = new Box<>();
Box rawBox = stringBox; // OK
```

No obstant les construccions anteriors no s'haurien d'utilitzar excepte en els casos on calgui aconseguir algun tipus de retrocompatibilitat amb codi anterior a Java 5 ja que el què estem fent és eliminar la seguretat de tipus que proporcionen els genèrics què és precisament pel que els volíem en primer lloc.

44.7. Tipus acotats (bounded types)

Fins ara hem vist que els paràmetres tipus poden ser substituïts per a qualsevol tipus de classe. A vegades és útil **limitar els tipus** que es poden passar com a paràmetre, vegem-ho amb un exemple:

Volem construir una classe genèrica amb un mètode que **retorna la mitjana aritmètica dels números d'una matriu de números**. El problema és que volem que la matriu pugui contenir enters i números en coma flotant indistintament.

El primer intent podria ser el següent:

```

class MathArray<T> {
    T[] nums;
    Utils(T[] o) {
        nums = o;
    }
}

```

```

}

// Retornem un double en tots els casos
double mitjana() {
    double sum = 0.0;
    for(int i=0; i < nums.length; i++)
        sum += nums[i].doubleValue(); // Error ①
    return sum / nums.length;
}
}
}

```

- ① El problema amb aquesta línia és que no totes les classes implementen el mètode **doubleValue()** i el compilador no té manera de saber si el tipus **T** implementa el mètode o no.

Però el que sí sabem nosaltres és que la classe **Number** declara aquest mètode i que tots els tipus numèrics deriven d'aquesta classe, i per tant implementen el mètode **doubleValue()**.

Per tant, si som capaços de limitar el paràmetre **T** a només tipus derivats de la classe **Number**, és a dir, limitar el paràmetre només a tipus numèrics, assegurem la existència de **doubleValue()** per a tota **T** possible_* i per tant el compilador deixarà de donar error.

Per fer-ho utilitzarem la paraula clau **extends** com en el següent exemple:

```

public class MathArray<T extends Number> {
    T[] nums;

    Estat(T[] o) {
        nums = o;
    }

    // Retornem un double en tots els casos
    public double mitjana() {
        double sum = 0.0;
        for (int i = 0; i < nums.length; i++) { ②
            sum += nums[i].doubleValue();
        }
        return sum / nums.length;
    }
}

```

```

    }
}

```

- ➊ Obliguem a que el paràmetre tipus **T** sigui un tipus derivat de **Number**.
- ➋ Com que **T** és un tipus derivat de **Number** segur que implementa el mètode **doubleValue()**.



La paraula clau **extends** aplicada a paràmetres tipus acotats no només aplica a herència de classes sinó que també s'utilitza per implementació d'interfícies.

Per exemple, la declaració `class <T extends I>` és correcte tant si **I** és una **classe** com si és una **interfície**.

44.7.1. Múltiples tipus acotats

És possible especificar més d'un tipus com una acotació, tenint en compte que només un d'ells pot ser una classe i els demés han de ser interfícies. Es faria de la següent manera:

```

Class A { /* ... */ }
interface B { /* ... */ }
interface C { /* ... */ }

class D <T extends A & B & C> { /* ... */ }

```



En l'exemple anterior cal que el primer tipus després de **extends** sigui la classe en cas contrari es produiria un error de compilació.

44.8. Arguments comodí

A vegades la seguretat de tipus proporcionada pels genèrics suposa un impediment en construccions perfectament acceptables.

Per exemple, considerem l'exemple anterior:

```
public class MathArray<T extends Number> {
```

```

T[] nums;

MathArray(T[] o) {
    nums = o;
}

// Retornem un double en tots els casos
public double mitjana() {
    double sum = 0.0;
    for (int i = 0; i < nums.length; i++) {
        sum += nums[i].doubleValue();
    }
    return sum / nums.length;
}
}

```

Suposem que volem afegir un mètode anomenat **mateixaMitjana(Estat: e)** que determina si dos objectes **Estat** contenen matrius amb valors amb la mateixa mitjana **independentment del tipus de dades numèriques que contingui cada matriu.**

És a dir, el comportament que volem aconseguir tenint en compte el codi següent és un resultat de **Les mitjanes aritmètiques són les mateixes:**

```

Integer inums[] = {1, 2, 3, 4, 5};
Double dnums[] = {1.0, 2.0, 3.0, 4.0, 5.0};
MathArray<Integer> iob = new MathArray<Integer>(inums);
MathArray<Double> dob = new MathArray<Double>(dnums);

if (iob.mateixaMitjana(dob)) {
    System.out.println("Les mitjanes aritmètiques són les mateixes");
} else {
    System.out.println("Les mitjanes aritmètiques són diferents.");
}

```

44.8.1. Primer intent de solució

Probablement el primer intent d'implementació del mètode seria similar a:

```

boolean mateixaMitjana(MathArray<T> ob) {
    if(this.mitjana() == ob.mitjana())
        return true;
}

```

```

    return false;
}

```

El codi anterior generaria un error de compilació a l'hora de cridar **mateixaMitjana()**. Vegem-ho:

```

Integer inums[] = {1, 2, 3, 4, 5};
Double dnums[] = {1.0, 2.0, 3.0, 4.0, 5.0};
MathArray<Integer> iob = new MathArray<Integer>(inums);
MathArray<Double> dob = new MathArray<Double>(dnums);

if (iob.mateixaMitjana(dob)) { ❶
    System.out.println("Les mitjanes aritmètiques són les mateixes");
} else {
    System.out.println("Les mitjanes aritmètiques són diferents.");
}

```

- ❶ La manera d'implementar el mètode **boolean mateixaMitjana(Estat<T> ob)** implica que tant l'objecte **iob** com l'objecte **dob** són del mateix tipus. **iob** és de tipus **Estat<Integer>**, **dob** és de tipus **Estat<Double>** i aquest és l'origen del problema.

44.8.2. Segon intent de solució

Una altra manera de veure-ho podria ser definir el mètode de la següent manera:

```

public boolean mateixaMitjana(MathArray ob) { ❶
    if (this.mitjana() == ob.mitjana()) {
        return true;
    }
    return false;
}

```

- ❶ Estem prescindint dels genèrics i per tant permetem que el paràmetre **ob** sigui de tipus **Estat** independentment del *paràmetres tipus*.



Tot i que aquesta solució funcionaria recordem que **no s'hauria d'utilitzar excepte per temes de retrocompatibilitat**, que no és el cas. Per tant **no donarem aquesta solució per bona!!**

44.8.3. Tercer intent de solució

La solució correcte emmarcada dins els genèrics és utilitzar el comodí ? com a *paràmetre tipus*. Vegem-ho:

```
public boolean mateixaMitjana(MathArray<?> ob) { ❶
    if (this.mitjana() == ob.mitjana()) {
        return true;
    }
    return false;
}
```

- ❶ El que estem dient aquí és que acceptem un objecte estat amb **qualsevol paràmetre tipus** que en definitiva és el que volíem.



Cal entendre que el comodí ? no afecta a quins tipus d'objectes **Estat** es poden crear això queda determinat per la clàusula **extends** a la declaració de la classe **Estat**.

El comodí fa referència únicament a qualsevol objecte **Estat** vàlid.

44.8.4. Codi d'exemple

```
package generics;

public class MathArray<T extends Number> {

    T[] nums;

    MathArray(T[] o) {
        nums = o;
    }

    // Retornem un double en tots els casos
    public double mitjana() {
        double sum = 0.0;
        for (int i = 0; i < nums.length; i++) {
            sum += nums[i].doubleValue();
        }
        return sum / nums.length;
    }
}
```

```

    }

    public boolean mateixaMitjana(MathArray<?> ob) {
        if (this.mitjana() == ob.mitjana()) {
            return true;
        }
        return false;
    }
}

```

```

package generics;

public class Main {

    public static void main(String[] args) {
        Integer inums[] = {1, 2, 3, 4, 5};
        Double dnums[] = {1.0, 2.0, 3.0, 4.0, 5.0};
        Float fnums[] = {1.1f, 2.2f, 3.3f, 4.4f, 5.5f};

        MathArray<Integer> iob = new MathArray<>(inums);
        MathArray<Double> dob = new MathArray<>(dnums);
        MathArray<Float> fob = new MathArray<>(fnums);

        printResults(iob, fob);
        printResults(iob, dob);
        printResults(dob, fob);
    }

    private static void printResults(MathArray<?> e1, MathArray<?> e2) {
        if (e1.mateixaMitjana(e2)) {
            System.out.println("Les mitjanes aritmètiques són les
mateixes");
        } else {
            System.out.println("Les mitjanes aritmètiques són
diferents.");
        }
    }
}

```

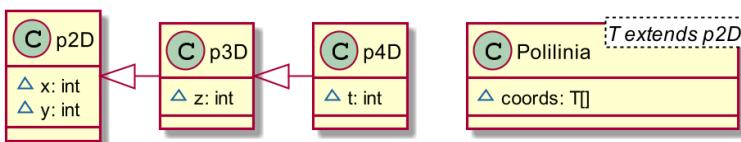
44.9. Arguments comodí acotats

Els arguments comodí es poden acotar de la mateixa manera que un *paràmetre tipus*.

Els *arguments comodí acotats* són especialment importants quan es treballa amb genèrics que operen sobre jerarquies de classes.

Utilitzarem el següent exemple per veure el seu funcionament:

Considerem la següent jerarquia de classes:



```

class p2D {

    private int x;
    private int y;

    public p2D(int a, int b) {
        x = a;
        y = b;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }
}

class p3D extends p2D {

    private int z;

    public p3D(int a, int b, int c) {
        super(a, b);
        z = c;
    }

    public int getZ() {
        return z;
    }
}
  
```

```

}

class p4D extends p3D {

    private int t;

    public p4D(int a, int b, int c, int d) {
        super(a, b, c);
        t = d;
    }

    public int getT() {
        return t;
    }
}

```

I la classe **Polilinia**:

```

class Polilinia<T extends p2D> {

    private T[] punts;

    public Polilinia(T[] o) {
        punts = o;
    }

    public int getNumPunts() {
        return punts.length;
    }

    public T getPunt(int index) {
        return punts[index];
    }
}

```

Volem escriure un mètode que **només** mostri les coordenades **x** i **y** de cada element de la matriu dins del objecte **Polilinia** donat.

Ho podríem fer de la següent manera:

```

class Polilinies {

    public static void mostrarXY(Polilinia<?> p) {

```

```

        System.out.println("Coordenades:");
        for (int i = 0; i < p.getNumPunts(); i++) {
            System.out.println("(" + p.getPunt(i).getX() + ", " +
p.getPunt(i).getY() + ")");
        }
        System.out.println();
    }
}

```

Fins aquí tot bé. Ara volem crear un mètode, **mostrarXYZ()**, que mostri les coordenades **x**, **y**, **z** de cada element de la matriu dins del objecte **Coord** donat.

El problema, en aquest cas, és que no tots els objectes **Coord** **tenen necessàriament una coordenada z**. Per tant, necessitem una manera d'escriure el mètode de manera que mostri les coordenades x,y i z per **Coord<p3D>** i **Coord<p4D>** i evitar que el mètode aparegui per **Coord<p2D>**.

La solució passa per acotar l'argument tipus comodí, un argument comodí acotat especifica una **cota superior** o bé una **cota inferior** per l'argument tipus. **Això permet restringir els tipus d'objectes sobre els quals pot operar un mètode**.

Vegem el següent mètode **mostrarXYZ()**

```

public static void mostrarXYZ(Polininia<? extends p3D> p) { ❶
    System.out.println("Coordenades:");
    for (int i = 0; i < p.getNumPunts(); i++) {
        System.out.println("(" + p.getPunt(i).getX() + ", " +
p.getPunt(i).getY()
        + ", " + p.getPunt(i).getZ() + ")");
    }
    System.out.println();
}

```

- ❶ Al afegir la paraula clau **extends** al comodí estem indicant que **admetem qualsevol tipus de dades que hereti (o implementi en el cas de les interfícies) p3D**. Com a conseqüència cridar al mètode amb un paràmetre de tipus **p2D** generarà un error en temps de compilació, garantint la seguretat de tipus, que és precisament el que volíem aconseguir.

Amb la construcció **? extends p3D** hem establert una **cota superior** a ? en el sentit que no permetem objectes de tipus per sobre de **p3D** en una jerarquia d'herència.

Seguint amb la mateixa idea també es poden definir **cotes inferiors** amb la construcció `? super classe`, en aquest cas només acceptaríem com a arguments objectes de classes per sobre de `classe` en la jerarquia d'objectes.

44.9.1. Codi d'exemple

```
package punts;

public class Punts {

    public static void main(String[] args) {
        p2D p2d[] = {
            new p2D(0, 0),
            new p2D(7, 9),
            new p2D(18, 4),
            new p2D(-1, -23)
        };
        Polilinia<p2D> p2dCoords = new Polilinia<p2D>(p2d);
        System.out.println("Contingut de p2dCoords.");
        Polilinies.mostrarXY(p2dCoords);
        // Polilinies.mostrarXYZ(tdlocs); // Error
        // Polilinies.mostrarXYZT(tdlocs); // Error

        p4D p4d[] = {
            new p4D(1, 2, 3, 4),
            new p4D(6, 8, 14, 8),
            new p4D(22, 9, 4, 9),
            new p4D(3, -2, -23, 17)
        };
        Polilinia<p4D> p4dCoords = new Polilinia<p4D>(p4d);
        System.out.println("Contingut de p4dCoords.");

        Polilinies.mostrarXY(p4dCoords);
        Polilinies.mostrarXYZ(p4dCoords);
        Polilinies.mostrarXYZT(p4dCoords);
    }

}

class p2D {

    private int x;
    private int y;
```

```
public p2D(int a, int b) {
    x = a;
    y = b;
}

public int getX() {
    return x;
}

public int getY() {
    return y;
}

class p3D extends p2D {

    private int z;

    public p3D(int a, int b, int c) {
        super(a, b);
        z = c;
    }

    public int getZ() {
        return z;
    }
}

class p4D extends p3D {

    private int t;

    public p4D(int a, int b, int c, int d) {
        super(a, b, c);
        t = d;
    }

    public int getT() {
        return t;
    }
}

class Polilinia<T extends p2D> {

    private T[] punts;
```

```
public Polilinia(T[] o) {
    punts = o;
}

public int getNumPunts() {
    return punts.length;
}

public T getPunt(int index) {
    return punts[index];
}

}

class Polilinies {

    public static void mostrarXY(Polilinia<?> p) {
        System.out.println("Coordenades:");
        for (int i = 0; i < p.getNumPunts(); i++) {
            System.out.println("(" + p.getPunt(i).getX() + ", " +
p.getPunt(i).getY() + ")");
        }
        System.out.println();
    }

    public static void mostrarXYZ(Polilinia<? extends p3D> p) {
        System.out.println("Coordenades:");
        for (int i = 0; i < p.getNumPunts(); i++) {
            System.out.println("(" + p.getPunt(i).getX() + ", " +
p.getPunt(i).getY() +
", " + p.getPunt(i).getZ() + ")");
        }
        System.out.println();
    }

    public static void mostrarXYZT(Polilinia<? extends p4D> p) {
        System.out.println("Coordenades:");
        for (int i = 0; i < p.getNumPunts(); i++) {
            System.out.println("(" + p.getPunt(i).getX() + ", " +
p.getPunt(i).getY() +
", " + p.getPunt(i).getZ() + ", " +
p.getPunt(i).getT() + ")");
        }
        System.out.println();
    }
}
```

{}

44.10. Mètodes genèrics

És possible declarar un mètode genèric que utilitzi un o més paràmetres tipus. De fet és possible crear un mètode genèric dins d'una classe no genèrica.

Vegem-ho amb un exemple, el següent programa declara una classe no genèrica i un mètode estàtic anomenat **pertany**. El mètode **pertany** determina si l'objecte passat com a primer paràmetre pertany a la matriu passada com a segon paràmetre.

```
public class MetodeGenericDemo {

    static <T extends Comparable<T>, V extends T> boolean pertany(T x,
        V[] y) { ❶
        for (int i = 0; i < y.length; i++) {
            if (x.equals(y[i])) {
                return true;
            }
        }
        return false;
    }

    public static void main(String[] args) {
        Integer nums[] = {1, 2, 3, 4, 5};
        if (pertany(2, nums)) { ❷
            System.out.println("2 pertany a nums");
        }
        if (!pertany(7, nums)) {
            System.out.println("7 no pertany a nums");
        }
        System.out.println();

        String cadenes[] = {"un", "dos", "tres", "quatre", "cinc"};
        if (pertany("dos", cadenes)) {
            System.out.println("dos pertany a cadenes");
        }
        if (!pertany("set", cadenes)) {
            System.out.println("set no pertany a cadenes");
        }
    }
}
```

```

        // Es genera un error ja que els tipus no són compatibles
        // if(pertany("dos", nums)) ❸
        // System.out.println("dos pertany a nums");
    }
}

```

- ❶ Els paràmetres tipus es declaren abans del tipus de retorn del mètode. El tipus de dades del primer paràmetre ha de poder-se comparar i per tant exigim que implementi la interfície **Comparable<T>**.
- ❷ El mètode **pertany** es crida sense especificar els arguments tipus, s'extrapolen automàticament.
Si fos necessari podríem explicitar els paràmetres tipus de la següent manera: <Integer, Integer>pertany(2, nums)
- ❸ Com que el primer argument és *String* i *Integer* no deriva del primer es genera un error en temps de compilació.

En definitiva, podem generalitzar la sintaxi dels mètodes genèrics de la següent manera:

```
<type-param-list > ret-type meth-name (param-list) { // ...
```

44.11. Constructors genèrics

És possible fer un constructor genèric encara que la classe no ho sigui, vegem-ho amb un exemple:

```

public class GenericConstructorDemo {

    public static void main(String[] args) {
        GenericConstructor test = new GenericConstructor(100);
        GenericConstructor test2 = new GenericConstructor(123.5F);
        test.showval();
        test2.showval();
    }
}

class GenericConstructor {

    private double valor;
}

```

```

<T extends Number> GenericConstructor(T arg) {
    valor = arg.doubleValue();
}

void showval() {
    System.out.println("valor: " + valor);
}
}

```

44.12. Interfícies genèriques

A més a més de classes i mètodes genèrics també es poden implementar interfícies genèriques.

Implementar una interfície genèrica és molt similar a implementar una classe genèrica, el següent exemple ho mostra:

```

interface MinMax<T extends Comparable<T>> {

    T min();

    T max();
}

class GenericInterficieDemo<T extends Comparable<T>> implements
MinMax<T> { ❶

    T[] valors;

    GenericInterficieDemo(T[] valors) {
        this.valors = valors;
    }

    public T min() {
        T v = valors[❷];
        for (int i = ❸; i < valors.length; i++) {
            if (valors[i].compareTo(v) < ❹) {
                v = valors[i];
            }
        }
        return v;
    }
}

```

```

public T max() {
    T v = valors[0];
    for (int i = 1; i < valors.length; i++) {
        if (valors[i].compareTo(v) > 0) {
            v = valors[i];
        }
    }
    return v;
}

public class Main {

    public static void main(String[] args) {
        Integer inums[] = {3, 6, 2, 8, 6};
        Character chs[] = {'b', 'r', 'p', 'w'};
        GenericInterficieDemo<Integer> iob = new
        GenericInterficieDemo<>(inums);
        GenericInterficieDemo<Character> cob = new
        GenericInterficieDemo<>(chs);

        System.out.println("Màxim valor de inums: " + iob.max());
        System.out.println("Minim valor de inums: " + iob.min());
        System.out.println("Màxim valor de chs: " + cob.max());
        System.out.println("Minim valor de chs: " + cob.min());
    }
}

```

- 1** L'acotació del paràmetre **T** apareix a la declaració de la classe **no** ha d'aparèixer al **implements** de la interfície. De fet, l'acotació ha de ser la mateixa i una es pot extrapolar de l'altra.
 Dit d'una altra manera, la següent declaració seria incorrecta i generaria un error: `class MyClass<T extends Comparable<T>> implements MinMax<T extends Comparable<T>>`



En general, si una classe implementa una interfície genèrica, la classe també ha de ser genèrica., com a mínim ha d'agafar el paràmetre tipus que es passa a la interfície.

Per exemple, la següent declaració és incorrecte:

```
class Classe implements MinMax<T> { ... }
```

Com que la classe no declara un paràmetre tipus no hi ha manera de passar-li un a la interfície **MinMax**.

Ara bé, si la classe implementa un tipus específic de interfície genèrica, no hi ha necessitat de que la classe rebi un paràmetre tipus i la declaració correcta seria la següent:

```
class Classe implements MinMax<Integer> { ... }
```

En definitiva, podem generalitzar la sintaxi de les interfícies genèriques de la següent manera:

```
interface interface-name<type-param-list> { // ... }
```

i quan s'implementa la interfície:

```
class class-name<type-param-list> implements interface-name<type-arg-list> { }
```

45

La interfície Comparable

Suposem que es vol implementar un algorisme d'ordenació. Per exemple un algorisme anomenat **ordenació per inserció**, útil quan les dades del vector ja estan bastant ordenades d'entrada.

Ordenació per inserció

L'ordenació per inserció (**insertion sort** en anglès) és una manera molt natural d'ordenar una col·lecció d'elements.

1. Inicialment es té un sol element, que obviament és un conjunt ordenat.
2. Després, quan hi ha **k** elements ordenats de menor a major, es pren l'element **k+1** i es compara amb tots els elements ja ordenats, parant quan es troba un element menor (tots els elements majors han estat desplaçats una posició a la dreta) o quan ja no es troben elements (tots els elements han estat desplaçats i aquest és el més petit).
3. En aquest punt s'insereix l'element **k+1**.

Requereix $O(n^2)$ operacions per a ordenar una llista de n elements.

El codi seria així:

```
public static void sort(int[] vector) {  
    int valorPivot;  
    int indexACanviar, pivot;
```

```

for (pivot = 1; pivot < vector.length; pivot++) {
    valorPivot = vector[pivot];
    indexACanviar = pivot - 1;
    while (indexACanviar >= 0 && vector[indexACanviar] > valorPivot) {
        vector[indexACanviar + 1] = vector[indexACanviar];
        indexACanviar = indexACanviar - 1;
    }
    vector[indexACanviar+1] = valorPivot;
}
}

```

Amb això hem aconseguit tenir un algorisme per ordenar vectors d'enters. Però, què passa si el nostre vector és de nombres reals? Hauríem de fer un altre mètode similar:

```

public static void sort(double[] vector) {
    double valorPivot;
    int indexACanviar, pivot;

    for (pivot = 1; pivot < vector.length; pivot++) {
        valorPivot = vector[pivot];
        indexACanviar = pivot - 1;
        while (indexACanviar >= 0 && vector[indexACanviar] > valorPivot) {
            vector[indexACanviar + 1] = vector[indexACanviar];
            indexACanviar = indexACanviar - 1;
        }
        vector[indexACanviar+1] = valorPivot;
    }
}

```

Fixeu-vos que els dos mètodes són idèntics, només ha canviat el tipus de dades de vector i el del **valorPivot**.

Ara suposem que tenim una classe **Data** que emmagatzema una determinada data a una determinada part del món.

Les dates es poden ordenar si tenim un bon criteri de quina data va abans que quina altra.

Per exemple, podríem passar les dates al mateix fus horari (UTC), comptar els segons que han passat des que va començar el 1970, i això ens donaria un nombre enter per cada data que seria senzill de comparar.

Si tenim un mètode **getTemps()** a la classe Data que ens retorna el nombre desitjat, l'algoritme quedaria així:

```
public static void sort(Data[] vector) {  
    Data valorPivot;  
    int indexACanviar, pivot;  
  
    for (pivot = 1; pivot < vector.length; pivot++) {  
        valorPivot = vector[pivot];  
        indexACanviar = pivot - 1;  
        while (indexACanviar >= 0 && vector[indexACanviar].getTemps() >  
            valorPivot.getTemps()) {  
            vector[indexACanviar + 1] = vector[indexACanviar];  
            indexACanviar = indexACanviar - 1;  
        }  
        vector[indexACanviar+1] = valorPivot;  
    }  
}
```

Arrivats a aquest punt hauríem de ser capaços de veure que hi ha alguna cosa que no estem dissenyant correctament, **no té massa sentit** que s'hagi de redefinir la funció **sort()** per a cada tipus de dades que volguem tractar tenint en compte que totes les versions de **sort()** són pràcticament iguals.

Primer de tot anem a resoldre el problema de la comparació, per obtenir un algorisme únic. Llavors tractarem el problema del tipus de dades.

Per fer-ho seguirem dues de les màximes de la programació orientada a objectes: **separa aquelles coses que varien de les que són constants** (ho farem encapsulant allò que varia) i fes que **cada classe sigui responsable de sí mateixa**.

En el nostre exemple, qui és responsable de saber com es comparen dues dates?

Naturalment la classe **Data**, no pas la classe **Algorisme**. Igualment, la classe **Integer** serà responsable de comparar nombres enters, i així successivament.

Aleshores, creem un mètode a la classe Data per comparar dates. Podem fer, per exemple, que aquest mètode **retorni un nombre negatiu si la primera data és menor que la segona, positiu si la segona és menor que la primera, i 0 si són idèntiques**. El codi quedaría:

```
public class Data {  
    . . .  
    public int compareTo(Data arg0) {  
        return this.getTemps() - arg0.getTemps();  
    }  
}
```

D'aquesta manera, el mètode d'ordenació seria:

```
public static void sort(Data[] vector) {  
    Data valorPivot;  
    int indexACanviar, pivot;  
  
    for (pivot = 1; pivot < vector.length; pivot++) {  
        valorPivot = vector[pivot];  
        indexACanviar = pivot - 1;  
        while (indexACanviar >= 0 &&  
               vector[indexACanviar].compareTo(valorPivot) > 0) {  
            vector[indexACanviar + 1] = vector[indexACanviar];  
            indexACanviar = indexACanviar - 1;  
        }  
        vector[indexACanviar+1] = pivot;  
    }  
}
```

En aquest punt tenim un algorisme que funcionaria per a qualsevol classe que tingués un mètode **compareTo** que segueixi aquestes regles, de manera que el codi seria idèntic tant si comparem dates, com enters, o qualsevol altre classe comparable.

Només queda arreglar el tema dels tipus. No podem utilitzar **Object** com a classe genèrica, perquè no tots els objecten tenen un mètode **compareTo**.

D'alguna manera hem de dir que *Integer*, **Double* i *Data* tenen quelcom en comú, i no ho podem fer per herència perquè no té sentit que una data derivi d'un nombre.

La solució és, evidentment, capturar el què tenen en comú totes aquestes classes en una interfície, i assegurar que totes les classes que implementin aquesta interfície tindran el mètode **compareTo**.

La interfície seria alguna cosa com:

```
public interface Comparable {  
    public int compareTo(Object arg0);  
}
```

I **Data**, **Integer** i **Double** implementarien aquesta interfície:

```
public class Data implements Comparable
```

El codi gairebé definitiu per l'algorisme d'ordenació seria:

```
public static void sort(Comparable[] vector) {  
    Comparable valorPivot;  
    int indexACanviar, pivot;  
  
    for (pivot = 1; pivot < vector.length; pivot++) {  
        valorPivot = vector[pivot];  
        indexACanviar = pivot - 1;  
        while (indexACanviar >= 0 &&  
               vector[indexACanviar].compareTo(valorPivot) > 0) {  
            vector[indexACanviar + 1] = vector[indexACanviar];  
            indexACanviar = indexACanviar - 1;  
        }  
        vector[indexACanviar+1] = pivot;  
    }  
}
```

El codi anterior ja funciona perfectament, però encara es pot millorar una mica.

El mètode rep un vector d'objectes que implementen tots ells la interfície **Comparable**. El problema és que aquest vector podria tenir objectes de diferents classes que no tingui sentit comparar entre ells. Per exemple, no té sentit comparar un enter amb una Data

Utilitzant les classes genèriques es pot especificar que les classes es poden comparar només amb algunes altres classes, però no amb qualsevol.

Fent-ho així, la interfície quedaria:

```
public interface Comparable<T> {  
    public int compareTo(T arg0);
```

```
}
```

La declaració de les classes:

```
public class Data implements Comparable<Data>
```

I el mètode d'ordenació:

```
public class Algoritmes {  
    public static <T extends Comparable<T>> void sort(T[] vector) {  
        T valorPivot;  
        int indexACanviar, pivot;  
  
        for (pivot = 1; pivot < vector.length; pivot++) {  
            valorPivot = vector[pivot];  
            indexACanviar = pivot - 1;  
            while (indexACanviar >= 0 &&  
                   vector[indexACanviar].compareTo(valorPivot) > 0) {  
                vector[indexACanviar + 1] = vector[indexACanviar];  
                indexACanviar = indexACanviar - 1;  
            }  
            vector[indexACanviar+1] = pivot;  
        }  
    }  
}
```

Fixeu-vos que en la declaració del mètode s'especifica que es pot rebre qualsevol tipus amb l'única condició que sigui comparable amb sí mateix, és a dir, que implementi Comparable<T>, on T és la seva classe.

Refinant encara una mica més, es pot declarar el mètode com:

```
public static <T extends Comparable<? super T>> void sort(T[] vector) {
```

Aquesta declaració indica que es pot rebre qualsevol tipus T amb la condició que es pugui comparar amb alguna altra classe que sigui o T o una classe de la qual T derivi.

Per exemple, si utilitzem la classe DataLloc, que especifica a més d'una data, una posició geogràfica concreta, el mateix mètode d'ordenació ens serviria per ordenar un vector d'objectes d'aquest tipus, comparant-los com a dates.

El que acabem de fer és crear la interfície **Comparable** del Java i un mètode similar al **sort** de la classe **Collections**.

La interfície Comparator

Si volem comparar objectes entre ells, a nivell de disseny, tenim dues opcions.

1. Considerem que els propis objectes tenen la capacitat de poder-se comparar amb d'altres objectes semblants. Aquesta és la estratègia que hem seguit amb la interfície **CompareTo**
2. Considerem que el fet de comparar dos objectes és una funcionalitat externa als objectes i és responsabilitat d'un tercer objecte comparar el objectes originals. Aquesta és la estratègia seguida amb la interfície **Comparator**.

Seguint el raonament de l'explicació anterior crearíem una interfície **Comparator** de la següent manera.

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```

Suposem, per exemple, que el que volem comparar són **Articles** pel seu pes.

```
public class Article {  
  
    private int id;  
    private double pes;  
  
    public Article(int id, double pes) {  
        this.id = id;  
        this.pes = pes;
```

```

}

public int getId() {
    return id;
}

public double getPes() {
    return pes;
}

public void setPes(double pes) {
    this.pes = pes;
}

}

```

Crearem els tantes classes que implementin la interfície anterior com diferents comparacions necessitem. En particular:

```

public class ArticlePesComparator implements Comparator<Article> {

    @Override
    public int compare(Article o1, Article o2) {
        if (o1.getPes() > o2.getPes()) {
            return 1;
        }
        if (o1.getPes() < o2.getPes()) {
            return -1;
        }
        return 0;
    }
}

```

I ara podríem implementar un mètode **sort** de la següent manera:

```

public class Algoritmes {

    public static <T> void sort(T[] vector, Comparator<? super T> comp)
    {
        T valorPivot;
        int indexACanviar, indexPivot;
    }
}

```

Comparator i Comparable per les mateixes classes

```
for (indexPivot = 1; indexPivot < vector.length; indexPivot++) {  
    valorPivot = vector[indexPivot];  
    indexACanviar = indexPivot - 1;  
    while (indexACanviar >= 0 &&  
comp.compare(vector[indexACanviar], valorPivot) > 0) {  
        vector[indexACanviar + 1] = vector[indexACanviar];  
        indexACanviar = indexACanviar - 1;  
    }  
    vector[indexACanviar + 1] = valorPivot;  
}  
}
```



La interfície **Comparator** i el mètode **Collections.sort** ja estan implementats a les llibreries de Java.

46.1. Comparator i Comparable per les mateixes classes

A vegades pot tenir sentit implementar la interfície **Comparable** en una classe i a més crear-ne diversos **Comparators**.

Imaginem per exemple la classe **VolReal** que representa un vol concret que surt d'un aeroport cap a un altre, amb una data concreta de sortida i una data concreta d'arribada. De vegades interessarà ordenar els vols segons el moment de la seva sortida, mentre que altres vegades serà segons la seva arribada.

En aquests casos podem fer que l'ordre donat pel **compareTo** sigui el més habitual, i afegir un altre tipus d'ordenació, creant un objecte de tipus **Comparator**:

```
/*  
 * Retorna un nombre negatiu si aquest vol surt amb anterioritat a arg0,  
 * un nombre positiu si surt amb posterioritat, o 0 si surten  
 * simultàniament.  
 */  
@Override  
public int compareTo(VolReal arg0) {  
    return sortida.compareTo(arg0.sortida);  
}
```

Comparator i Comparable
per les mateixes classes

```
/*
 * Comparador per comparar dos vols segons el seu moment d'arribada.
 * Retorna positiu si arg0 arriba abans que arg1, negatiu si arriba
 * després,
 * o 0 si arriben simultàniament.
 */
public static final Comparator<VolReal> ORDRE_ARRIBADA =
    new Comparator<VolReal> () {
        @Override
        public int compare(VolReal arg0, VolReal arg1) {
            return arg0.getArribada().compareTo(arg1.getArribada());
        }
   };
```

En aquest exemple, hem creat una classe que implementa Comparator de forma implícita, és a dir, sense posar-li nom, i n'hem creat un únic objecte.

El mètode sort de Collections, per exemple, accepta un Comparator com a paràmetre, si volem utilitzar un tipus d'ordre diferent de l'ordre habitual de la classe:

```
Collections.sort(volsReals, VolReal.ORDRE_ARRIBADA);
```

47

Col·leccions

Una **col·lecció** és un objecte que agrupa múltiples elements en una sola unitat. En general poden permetre **emmagatzemar**, **recuperar**, **manipular** i **filtrar** dades.

Java disposa d'un "framework" específic per a col·leccions, el **Java Collections Framework**, es tracta d'una arquitectura unificada pel tractament de col·leccions.

Dins del Java Collections Framework hi podem trobar:

Interfícies

Els tipus abstractes de dades que representen col·leccions. Les interfícies permeten manipular les col·leccions **independentment dels detalls de la seva implementació**.

Implementacions

Es tracta d'implementacions concretes de les interfícies de les col·leccions.

Algoritmes

Són els **mètodes** que realitzen càlculs útils com per exemple **ordenar**, o **buscar** dins de les col·leccions. Es tracta d'algorismes polimòrfics, és a dir, el mateix mètode es pot utilitzar amb diferents implementacions de la interfície apropiada de la col·lecció.

47.1. Interfícies vs implementacions

La separació de la interfície d'una classe de la seva implementació és el principi principal darrera la encapsulació i l'amagament de dades assegurant que només el propi objecte controla les seves pròpies accions.

La declaració d'una classe combina la definició de la seva interfície externa (el conjunt dels seus mètodes no privats) amb una implementació d'aquesta interfície (el codi que du a terme el comportament d'aquests mètodes).

- La **interfície** defineix **què és el que un objecte pot fer** però no té la responsabilitat de fer-ho.
- La **implementació** du a terme la operació declarada a la interfície.
- El disseny de la classe especifica les interfícies que permeten a un objecte ser instantiat i operat degudament.
- La interfície ha de descriure completament com han d'interactuar amb la classe els usuaris d'aquesta.
- El comportament d'un objecte ha de ser invocat amb un missatge utilitzant una de les interfícies proporcionades.
- Els atributs i el comportament d'un objecte es controla enviant missatges al objecte.
- Unmissatge consisteix en el **nom de l'objecte receptor, el comportament desitjat que es vol utilitzar i tots els paràmetres requerits pel bon funcionament d'aquest comportament**.

47.2. Programar per la interfície, no per la implementació

Un dels principis fonamentals de la POO consisteix en **programar per la interfície, no per la implementació**. En el cas concret de les col·leccions aquest principi vol dir que al nostre codi no hauria de manipular implementacions concretes de les col·leccions sinó **sempre hauríem de manipular la interfície** de la col·lecció.

Per exemple, enllloc de fer:

```
ArrayList a = new ArrayList();
```

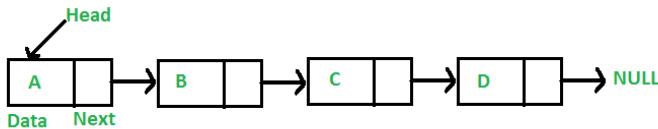
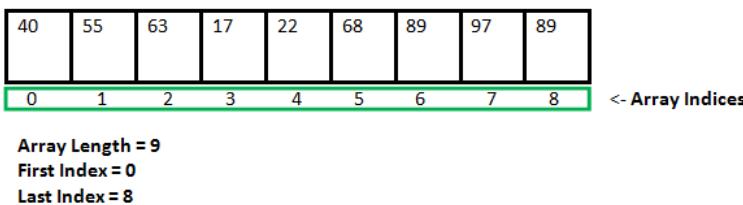
hauríem de fer:

```
List a = new ArrayList();
```

Estem dient que volem una estructura de dades que essencialment implementi **add**, **remove** i **get**, això és **List**. I que triem la implementació concreta de **ArrayList** però que si convingues podríem triar una implementació diferent, una **LinkedList** per exemple.

Exemple 47.1. ArrayList o LinkedList

ArrayList i **LinkedList** a nivell d'interfície són idèntiques, la diferència és a nivell d'implementació, internament una **ArrayList** és una **matriu** cosa que proporciona accés en temps constant, en canvi una **LinkedList** és una doble llista enllaçada, que proporciona *inserció* i *eliminació* d'elements en temps constant però no l'accés que varia en funció de la posició de l'element al que es vol accedir.



47.3. Un exemple, una interfície, dues implementacions

En el següent codi podeu veure dues implementacions de l'estructura **Pila**.

Primer s'ha definit una excepció personalitzada per al cas que la pila sigui buida quan es volen extreure elements. En segon lloc s'ha definit la interfície que ha de tenir una pila.

Finalment, s'han creat dues implementacions, una que utilitza un array, i una altra que utilitza una sèrie de nodes enllaçats.

47.3.1. La interfície

Interfície Pila.

```
package piles;

public interface Pila {

    void push(int n);

    int pop() throws PilaBuidaException;

    int peek() throws PilaBuidaException;

    boolean isEmpty();

}
```

47.3.2. Les dues implementacions

Implementació amb una matriu.

```
package piles;

public class PilaArray implements Pila {
    private int[] array = new int[10];
    private int mida = 0;

    private void duplicaArray() {
        int[] nouArray = new int[array.length*2];
        for (int i=0; i<array.length; i++) {
            nouArray[i] = array[i];
        }
        array = nouArray;
    }

    @Override
    public void push(int n) {
        if (mida==array.length) {
            duplicaArray();
        }
        array[mida] = n;
        mida++;
    }
}
```

```
@Override  
public int pop() throws PilaBuidaException {  
    if (isEmpty()) {  
        throw new PilaBuidaException("La pila és buida.");  
    }  
    mida--;  
    return array[mida];  
}  
  
@Override  
public int peek() throws PilaBuidaException {  
    if (isEmpty()) {  
        throw new PilaBuidaException("La pila és buida.");  
    }  
    return array[mida-1];  
}  
  
@Override  
public boolean isEmpty() {  
    return mida<=0;  
}  
}
```

Implementació amb una llista enllaçada.

```
package piles;  
  
public class PilaNode implements Pila {  
    private Node top;  
  
    @Override  
    public void push(int n) {  
        Node nouNode = new Node();  
        nouNode.setN(n);  
        nouNode.setPrev(top);  
        top = nouNode;  
    }  
  
    @Override  
    public int pop() throws PilaBuidaException {  
        if (isEmpty()) {  
            throw new PilaBuidaException("La pila és buida.");  
        }  
    }
```

```

int retorn = top.getN();
top = top.getPrev();
return retorn;
}

@Override
public int peek() throws PilaBuidaException {
    if (isEmpty()) {
        throw new PilaBuidaException("La pila és buida.");
    }
    return top.getN();
}

@Override
public boolean isEmpty() {
    return top==null;
}

private class Node {
    private int n;
    private Node prev;

    public int getN() {
        return n;
    }
    public void setN(int n) {
        this.n = n;
    }
    public Node getPrev() {
        return prev;
    }
    public void setPrev(Node prev) {
        this.prev = prev;
    }
}
}

```

Excepció personalitzada.

```

package piles;

public class PilaBuidaException extends Exception {
    private static final long serialVersionUID = 1L;

    public PilaBuidaException() {

```

```
super();
}

public PilaBuidaException(String message, Throwable cause, boolean enableSuppression, boolean writableStackTrace) {
    super(message, cause, enableSuppression, writableStackTrace);
}

public PilaBuidaException(String message, Throwable cause) {
    super(message, cause);
}

public PilaBuidaException(String message) {
    super(message);
}

public PilaBuidaException(Throwable cause) {
    super(cause);
}
}
```

47.3.3. Les proves

```
package main;

import piles.PilaNode;
import piles.Pila;
import piles.PilaArray;
import piles.PilaBuidaException;

public class Main {

    public static void main(String[] args) {
        System.out.println("Comprovant PilaNode...");
        Pila p = new PilaNode();
        comprovaPila(p);
        System.out.println("Comprovant PilaArray...");
        p = new PilaArray();
        comprovaPila(p);
    }

    public static void comprovaPila(Pila p) {
        for (int i=0; i<1000; i++) {
            p.push(i);
        }
    }
}
```

```
}

try {
    while (!p.isEmpty()) {
        System.out.println(p.pop());
    }
} catch (PilaBuidaException e) {
    System.out.println("fail");
}

try {
    p.pop();
    System.out.println("fail");
} catch (PilaBuidaException e) {
    System.out.println("ok");
}
}

public static void buidaPila(Pila p) {
    try {
        while (!p.isEmpty()) {
            System.out.println(p.pop());
        }
    } catch (PilaBuidaException e) {
        System.out.println("fail");
    }
}
}
```

47.4. Interfícies

Al següent diagrama es mostren les interfícies del Framework de col·leccions de Java.

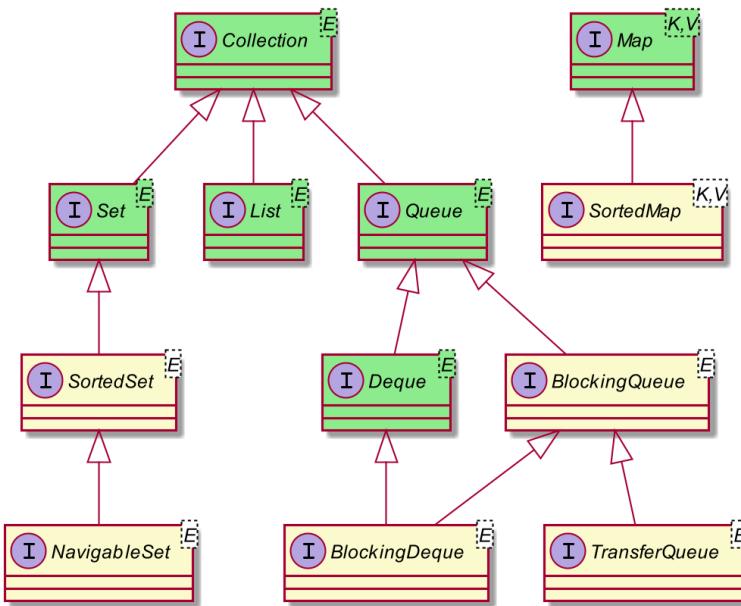


Figura 47.1. Interfícies principals de les col·leccions



La jerarquia anterior consisteix en dos arbres diferents, la raó és que un **Map** no és realment una col·lecció.

47.4.1. La interfície `java.util.Collection`

```
public interface Collection<E>
extends Iterable<E>
```

És l'arrel de tota la jerarquia de col·leccions, i per tant aquesta interfície descriu les operacions bàsiques que suporten la major part de col·leccions.

La plataforma Java **no proporciona implementació directa d'aquesta interfície**, proporciona implementacions d'interfícies més específiques com ara les interfícies **Set** o **List**.

El següent diagrama mostra la llista d'operacions suportades per la interfície:

I	Collection
<ul style="list-style-type: none"> ● add(E e): boolean ● addAll(Collection<? extends E> c): boolean ● clear(): void ● contains(Object o): boolean ● containsAll(Collection<?> c): boolean ● equals(Object o): boolean ● hashCode(): int ● isEmpty(): boolean ● iterator(): Iterator<E> ● parallelStream(): default Stream<E> ● remove(Object o): boolean ● removeAll(Collection<?> c): boolean ● removeIf(Predicate<? super E> filter): default boolean ● retainAll(Collection<?> c): boolean ● size(): int ● spliterator(): default Spliterator<E> ● stream(): default Stream<E> ● toArray(): Object[] ● toArray(T[] a): <T> T[] 	



Algunes col·leccions permeten elements duplicats i d'altres no i algunes estan ordenades.

47.4.2. Interfície java.util.Queue (Cua)

Les cues s'utilitzen en aquelles situacions en què no es poden processar tots els elements que ens arriben prou ràpidament, i se'ls posa en espera. Per exemple, la cua d'impressió de documents per una impressora permet que hi hagi diversos documents a l'espera de ser impresos, i que es processin en el mateix ordre com han anat arribant.

A part de les operacions comunes per les col·leccions, les cues proporcionen operacions d'**inserció, extracció i inspecció** addicionals.

En Java, la interfície **Queue** declara tots aquests mètodes, a més dels que ja hi havia a **Collection**. Cadascun dels mètodes hi és per duplicat:

- la versió que, si una operació no es pot realitzar (**add, remove, element**), llança una excepció.

- i la versió que retorna un valor especial (**offer**, **poll**, **peek**).

Normalment, però no sempre, són estructures FIFO (First In First Out). Amb les excepcions es troben les **cues amb prioritat**, on els elements s'ordenen segons una relació d'ordre, un "comparator" en Java.

Independentment de la ordenació utilitzada, el primer element de la cua, el **head**, és l'element que s'eliminarà quan es cridi al mètode **remove()**.

En una cua FIFO tots els nous elements són inserits al final de la cua, **tail**, però altres tipus de cues poden utilitzar diferents regles de col·locació.

L'operació d'inserció a una pila o a una cua s'anomena **push**, mentre que l'operació d'extracció s'anomena **pop**.

Taula 47.1. Mètodes de Queue

	Llença una excepció	Retorna un valor especial
Inserir	add(e)	offer(e)
Extreure	remove()	poll()
Examinar	element()	peek()

47.4.3. Interfície java.util.Deque

Deque és l'acrònim de **double ended queue**, representa una cua a la que se li poden afegir o treure elements pels dos costats.

Es solen utilitzar com a estructures **FIFO** o com a estructures **LIFO** (Last In First Out)

La interfície **Deque** és un subtipus de la interfície **Queue**, això implica que es pot utilitzar una **Deque** en totes les situacions on es pot utilitzar una **Queue**.



Com podem veure, en una pila (o una cua) no es pot accedir a qualsevol element de la seqüència. Per poder accedir a l'element 4, primer cal haver extret l'element 5.



A les API de Java hi ha la classe **Stack** que implementa una pila. Aquesta classe, però, prové de les primeres versions

del llenguatge, i actualment no es recomana el seu ús. En comptes d'això, es recomana utilitzar una implementació de la interfície **Deque**,

Els mètodes de **Deque** es resumeixen a la taula següent, segons el tipus d'operació i de si actuen sobre un extrem o l'altre:

Taula 47.2. Mètodes de Deque

	Primer element (head)		Últim element (tail)	
	Llença excepció	Valor especial	Llença excepció	Valor especial
Inserir	addFirst	offerFirst	addLast, add	offerLast, offer
Extreure	removeFirst, remove	pollFirst, poll	removeLast	pollLast
Examinar	getFirst, element	peekFirst, peek	getLast	peekLast

Com que aquesta interfície hereda de la interfície **Queue** quan una **Deque** s'utilitza com una **Queue** (FIFO) els mètodes heredats d'ela interfície **Queue** són equivalents als mètodes de **Deque** de la següent manera:

Taula 47.3. Comparació dels mètodes de Queue i Deque

Mètode a Queue	Mètode equivalent a Deque
add(e)	addLast(e)
offer(e)	offerLast(e)
remove()	removeFirst()
poll()	pollFirst()
element()	getFirst()
peek()	peekFirst()

47.4.4. Interfície java.util.List (Llista)

Una **Llista** és una seqüència d'elements en la qual es pot **inserir o extreure un element a qualsevol posició, i es pot accedir a qualsevol dels seus elements.**

Una llista és molt similar a un vector, amb l'excepció que canvia la seva mida quan s'afegeixen o s'extreuen elements.



És important no confondre el concepte de llista amb la implementació de llista enllaçada. La llista és una estructura dinàmica amb unes certes característiques i que permet una sèrie d'operacions, a nivell conceptual, i que es pot implementar de diverses maneres: com a llista enllaçada o com a vector dinàmic.

En Java, el concepte d'una llista ve descrit a la interfície **List** (del paquet **java.util**), no confondre amb la classe **List** de **java.awt** que és un control gràfic).

A més de les operacions de **Collection**, una **List** tindrà operacions per afegir, extreure, obtenir i intercanviar elements (**add**, **remove**, **get**, **set**) especificant la posició de l'element afectat, i operacions per obtenir la posició que ocupa un element dins de la llista (**indexOf** i **lastIndexOf**), entre d'altres.

Taula 47.4. Mètodes de List

	Mètode
Inserir	<code>add(int index, E element)</code>
Extreure	<code>remove(int index)</code>
Examinar	<code>get(int index)</code>
Reemplaçar	<code>set(int index, E element)</code>
Cercar	<code>indexOf(Object o)</code>

47.4.5. Interfície java.util.Set (Conjunt)

Un **conjunt** és una estructura dinàmica que conté una sèrie d'elements **no repetits i desordenats**, és a dir, els elements d'un conjunt no ocupen una posició o índex concret dins del conjunt, sinó que diem que un element és al conjunt o no és al conjunt.

En Java, els conjunts es defineixen a la interfície **Set**, amb mètodes com **add** per afegir elements, **remove** per extreure'ls, o **contains** per comprovar si un element és al conjunt o no. Els elements d'un conjunt també es poden recórrer amb un iterador o un bucle for-each, però no podem saber en quin ordre es farà aquest recorregut, només que es passarà un cop per cada element.

Els conjunt es defineixen a la interfície **Set**, i la seva implementació més genèrica és fa a la classe **HashSet**. També existeixen la interfície **SortedSet** i la classe **TreeSet**, que funcionen com conjunts en el sentit que els elements no es poden repetir, però que a més mantenen els elements ordenats segons el seu ordre natural (per exemple, un conjunt ordenat d'enters ens retornaria els elements ordenats de menor a major).

47.4.6. Interfície java.util.Map (Diccionari)

Un **diccionari**, també anomenat **array associatiu** o **mapa**, és una estructura dinàmica composta d'una col·lecció de **claus** úniques i una col·lecció de **valors** tals que a cada clau s'associa un valor.

Els diccionaris són una generalització del concepte de vector. Podem veure un vector com un diccionari que associa els primers enters a una sèrie de valors. En el cas dels diccionaris les claus no tenen perquè ésser enters consecutius, ni tan sols han d'ésser enters. Les **claus han de ser úniques** i apuntar a un únic valor (però res impedeix que aquest valor pugui ser una llista o qualsevol altra estructura).

Les següents interfícies deriven de Map:

SortedMap

SortedMap estén **Map** i manté les claus que s'hi introdueixen ordenades.

NavigableMap

NavigableMap estén **Map** i ofereix mètodes que permeten localitzar les claus més properes a un valor donat.

Exemples de diccionaris:

- Una agenda que associa a cada nom un telèfon. Els noms són les claus i els telèfons són els valors.
- Un diccionari de paraules: a cada paraula s'associa la seva definició. Les paraules són les claus i les definicions els valors.
- Una colla de països associats a la seva capital. Els noms dels països són les claus i els noms de les capitals els valors.
- Funcions matemàtiques amb un nombre finit de termes. Per exemple, el factorial des de l'1 fins al 100.

47.5. Implementacions

47.5.1. Implementacions de `java.util.Queue`

Les implementacions disponibles per a la interfície `java.util.Queue` són les següents:

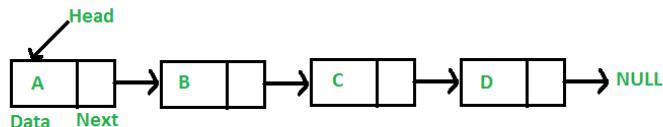
```
java.util.LinkedList
java.util.PriorityQueue
```



També hi ha implementacions sincronitzades de Queue al paquet `java.util.concurrent`.

LinkedList

Una llista enllaçada és una estructura de dades lineal en la que els elements no s'emmagatzem en ubicacions contigües de memòria. Els elements dins d'una llista enllaçada es vinculen utilitzant referències com mostra la imatge següent:

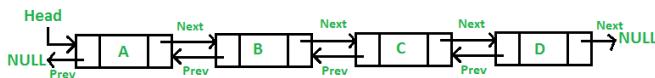


```
public class LinkedList {
    Node head;

    class Node {
        int data;
        Node next;

        Node(int d) {
            data = d;
        }
    }
}
```

La implementació proporcionada per Java és en particular la d'una **llista doblement enllaçada** com es mostra a la següent imatge:



```

public class DoubleLinkedList {
    Node head;

    class Node {
        int data;
        Node prev;
        Node next;

        Node(int d) {
            data = d;
        }
    }
}
  
```

La implementació principal de Java de les llistes doblament enllaçades es diu **LinkedList**. Una **LinkedList** implementa tant **List** com **Deque**, així que es pot utilitzar com una pila, una cua, o una llista.

PriorityQueue

PriorityQueue emmagatzema internament els seus elements d'acord amb el seu ordre natural (si implementen Comparable) o segons un **Comparator** passat a PriorityQueue .

47.5.2. Implementacions de java.util.Deque

Les implementacions disponibles per a la interfície **java.util.Deque** són les següents:

```

java.util.LinkedList
java.util.ArrayDeque
  
```

ArrayDeque

Representa una **Deque** implementada amb una matriu.

47.5.3. Implementacions de java.util.List

Les implementacions disponibles per a la interfície **java.util.List** són les següents:

```
java.util.ArrayList  
java.util.LinkedList  
java.util.Vector (IMPLEMENTACIÓ OBSOLETA)  
java.util.Stack (IMPLEMENTACIÓ OBSOLETA)
```

47.5.4. Implementacions de java.util.Set

Les implementacions disponibles per a la interfície **java.util.Set** són les següents:

```
java.util.EnumSet  
java.util.HashSet  
java.util.LinkedHashSet  
java.util.TreeSet
```

Cadascuna d'aquestes implementacions de **Set** es comporta una mica diferent respecte a l'ordre dels elements quan s'itera el conjunt i respecte el temps que es triga a inserir i accedir als seus elements.

EnumSet

Es una implementació especialitzada de **Set** per ser utilitzada en tipus **enum**. Internament està representat com un vector de bits proporcionant una implementació extraordinàriament compacta i eficient.

HashSet

HashSet està recolzat per un **HashMap**. No garanteix la seqüència dels elements quan s'iteren.

LinkedHashSet

LinkedHashSet difereix de **HashSet** garantint que l'ordre dels elements durant la iteració és el mateix que l'ordre en què es van inserir. El fet de tornar a inserir un element que ja es troba al **LinkedHashSet** no canvia aquest ordre.

TreeSet

TreeSet també garanteix l'ordre dels elements quan s'utilitzen, però l'ordre és l'ordre d'ordenació dels elements. És a dir, l'ordre en què es podrien ordenar

els elements si es va utilitzar una **Collections.sort()** en una **List** o matriu que conté aquests elements. Aquest ordre es determina o bé pel seu ordre natural (si implementen **Comparable**) o per una implementació específica del **Comparator**.



També hi ha implementacions de **Set** al paquet `java.util.concurrent`.

47.5.5. Implementacions de java.util.Map

L'API Java Collections conté les implementacions de Map següents:

```
java.util.HashMap
java.util.Hashtable
java.util.EnumMap
java.util.IdentityHashMap
java.util.LinkedHashMap
java.util.Properties
java.util.TreeMap
java.util.WeakHashMap
```

Les implementacions de Map més utilitzades són **HashMap** i **TreeMap**.

Cadascuna d'aquestes implementacions de **Map** es comporta una mica diferent respecte a l'ordre dels elements quan s'itera el conjunt i respecte el temps que es triga a inserir i accedir als seus elements.

HashMap

HashMap assigna una clau i un valor. No garanteix cap ordre dels elements emmagatzemats internament al mapa.

TreeMap

TreeMap també assigna una clau i un valor. A més, garanteix l'ordre en què s'atenen les claus o valors, que és l'ordre d'ordenació de les claus o valors.

Map no hereta de **Collection**, així que no tenim les operacions habituals de les altres col·leccions, en particular, no tenim iteradors per recórrer directament tot el contingut del diccionari.

Definir un nou diccionari

Per definir un diccionari cal especificar de quin tipus seran les claus i de quin tipus els valors. Per exemple:

```
Map<String, Integer> d = new HashMap<String, Integer>();
```

serà un diccionari que tindrà cadenes com a claus i enters com a valors.

- Per afegir elements:

```
d.put("Altura", 10);
d.put("Amplada", 5);
d.put("Fondària", 3);
```

O si tenim els elements en dos vectors, podríem fer:

```
for (i=0; i<clausIniciais.length; i++)
    d.put(clausIniciais[i], valorsIniciais[i]);
```

A partir d'un diccionari, crear-ne un altre independent (i possiblement de tipus diferent):

```
SortedMap<String, Integer> d2 = new TreeMap<String, Integer>(d);
```

Això crea un diccionari amb els mateixos valors que el primer diccionari, però que és independent, en el sentit que si modifiquem el diccionari original no es modifica la còpia.

Operacions sobre diccionaris

Per **afegir** una nova parella de clau-valor, simplement:

```
d.put("Ciutat", "Sabadell");
```

A un diccionari no hi poden haver claus repetides. Cal anar amb compte, perquè si ciutat ja hagués existit hauríem sobreescrit el seu valor.

Per **esborrar una clau** d'un diccionari:

```
d.remove("Ciutat");
```

Aquest mètode, a més, retorna el valor associat a la clau eliminada. Si el diccionari no conté la clau especificada, es retorna **null**.

Per **esborrar** totes les parelles de clau-valor del diccionari:

```
d.clear();
```

Per **comprovar** si **una clau** ja existeix al diccionari:

```
d.containsKey("Ciutat");
```

Retornarà **True** si la clau és al diccionari **d**.

Per **obtenir el valor** associat a una clau:

```
d.get(clau);
```

Ens retorna el valor associat a una clau i si no la troba, retorna **null**.

Per saber la **quantitat d'elements** que té el diccionari:

```
d.size();
```

Per saber **totes les claus o valors** que hi ha en un diccionari:

`d.keySet();` retorna un conjunt amb totes les claus que hi ha. L'ús d'aquest conjunt és molt interessant, perquè es manté actualitzat amb els canvis que es fan al diccionari, i el diccionari també incorpora les supressions que fem des del conjunt.

`d.values();` retorna una col·lecció amb tots els valors que hi ha al diccionari. Igual que en el cas anterior, els canvis que es realitzin sobre el diccionari es reflecteixen automàticament a la col·lecció que hem obtingut.

Exercici: per què **keySet** retorna un conjunt i **values** retorna una col·lecció?

Per **recórrer** tots els elements d'un diccionari:

```
Set<String> claus = d.keySet();
for (String clau : claus)
    System.out.println(clau+": "+d.get(clau));
```

O també:

```
Set<Entry<String, Integer>> info = d.entrySet();
for (Entry<String, Integer> dada : info)
    System.out.println(dada.getKey()+" "+dada.getValue());
```

47.6. Recórrer col·leccions

Hi ha tres maneres de recórrer una col·lecció:

1. Utilitzant operacions sobre un flux i expressions lambda.
De moment ho deixarem per més endavant...
2. Amb la construcció **for-each** o **enhaced for**..
3. Utilitzant *iteradors*.

47.7. Iteradors

D'entre els mètodes de **Collection** i **List**, el mètode **iterator()** mereix especial atenció. Imaginem que hem implementat la llista en dues classes, una com a vector dinàmic i l'altra com a llista enllaçada. Una de les operacions habituals és recórrer tots els elements de la llista.

En el cas del vector dinàmic, per fer aquest recorregut necessitem un índex, la posició del vector, que és un nombre enter, i podem accedir amb temps constant a qualsevol posició. En canvi, en la llista enllaçada l'índex es converteix en una referència a un **NodeLlista**, i és important recordar el punt on ens trobem, perquè accedir al següent element és ràpid, però accedir directament a un element intermedi del qual només sabem que ocupa la posició 100, per exemple, és lent.

Així, per poder recórrer la llista hem de conèixer la seva implementació interna, i hem d'accendir a una classe que en principi estava oculta com el **NodeLlista**.

Fent-ho d'aquesta manera, destruïm l'intent d'encapsulació que hem fet amb les piles i cues, i ja no serà possible escriure codi que sigui independent del tipus d'implementació que s'estigui utilitzant.

Per evitar això, a les biblioteques del Java s'ha creat la interfície **Iterator**. Quan vulguem recórrer una llista implementada com un vector, l'**Iterator** encapsularà un enter, que guardarà la posició de l'últim element visitat. Quan recorrem una llista enllaçada, l'**Iterator** encapsularà una referència al **NodeLlista** que s'hagi visitat. En el nostre programa, no accedirem directament ni a l'índex ni al **NodeLlista**, perquè ho farem sempre a través dels mètodes descrits a la interfície **Iterator**.

Realment, existiran dues classes que implementen **Iterator**, una per als **ArrayList** (diem-li **ArrayListIterator**) i una altra per a les **LinkedList** (diem-li **LinkedListIterator**). Però gràcies al polimorfisme, de la mateixa manera que treballàvem amb una **Pila** sense saber quin tipus de pila era, no sabrem de quina classe en concret és el nostre **Iterator**, només que és un **Iterator**. És per això que m'he inventat els noms de les classes dels iteradors, perquè realment no són classes públiques i per saber-los s'hauria d'anar a cercar en el codi font del Java.

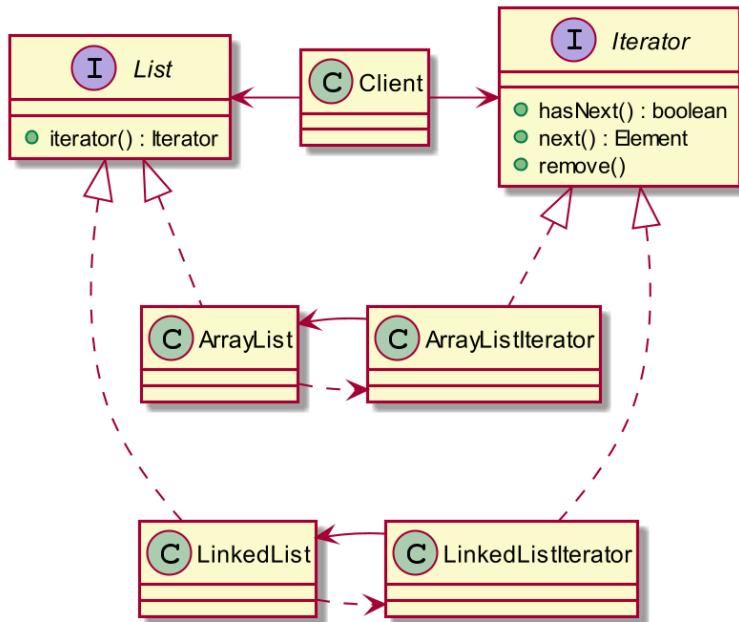
Queda una qüestió per resoldre. Quan hem implementat les piles, només hi havia un punt en què s'havia de conèixer amb quin tipus de pila volíem treballar i era en el moment de crear l'objecte. Per evitar això, i per evitar que puguem crear un **LinkedListIterator** per intentar iterar sobre un vector dinàmic, els constructors d'aquestes classes tampoc són públics. Com creo doncs un iterador per recórrer una llista? Doncs això és justament el que fa el mètode **iterator()** declarat a la interfície **Collection** (de fet, està declarat a **Iterable** i **Collection** amplia **Iterable**).

Resumint, el mètode **iterator()** crea un **Iterator** adequat per poder recórrer els elements d'una col·lecció.



S'aconsella no utilitzar la interfície **Enumerator**.

Fixeu-vos que un **Iterator** és similar al concepte de cursor en altres llenguatges, com el PL/SQL o el PHP.



Els mètodes com **iterator()**, que tenen com a objectiu construir un objecte adequat a la situació, sense que el programa que ho utilitza (el **Client**), hagi de saber abans el tipus adequat, s'anomenen **mètodes fàbrica (Factory Method)**. Fixeu-vos que són les classes que implementen **Iterable** les que decideixen de quin tipus serà l'iterador, en comptes de decidir-se directament a la interfície **Iterable**, o a **Client**.

L'esquema sencer utilitzat per amagar la implementació interna dels iteradors, i que es mostra a l'anterior diagrama UML, s'anomena **patró iterador (Iterator pattern)**.

Vegem els avantatges d'utilitzar un iterador amb alguns exemples de codi. El següent mètode rep una llista d'enters i troba l'enter més gran que hi ha:

```

public static int max(List<Integer> llista) {
    int i;
    int maxim = 0;

    if (llista.size() == 0) {
        throw new NoSuchElementException();
    }
  
```

```

maxim = llista.get(0);

for (i=1; i<llista.size(); i++) {
    if (llista.get(i) > maxim)
        maxim = llista.get(i);
}

return maxim;
}

```

Fixeu-vos que ens interessa que el mètode rebi **List** en comptes d'**ArrayList** o **LinkedList**: d'aquesta manera es pot utilitzar el mateix mètode per trobar el màxim dels dos tipus de llistes.

El problema és que si es tracta d'una **LinkedList**, i té molts elements, amb el mètode **get** tardarem molt a iterar, perquè cada vegada es recorren tots els elements fins a trobar l'element cercat.

Podem canviar el nostre mètode per tal que utilitzi iteradors:

```

public static int max(List<Integer> llista) {
    Iterator<Integer> it;
    int maxim = 0;
    int n;

    if (llista.size() == 0) {
        throw new NoSuchElementException();
    }

    maxim = llista.get(0);
    it = llista.iterator();
    while (it.hasNext()) {
        n = it.next();
        if (n > maxim)
            maxim = n;
    }

    return maxim;
}

```

Aquest mètode fa el mateix que l'anterior, però gràcies a l'ús de l'iterador, serà tant ràpid sobre un **ArrayList** com sobre un **LinkedList**. Fixeu-vos com els

iteradors també es parametritzén amb el tipus de dada que retornaran, utilitzant els genèrics.

És important notar que la crida al mètode **next()** ens retorna l'element següent i fa avançar l'iterador a la següent posició. Una sentència com:

```
if (it.next() > maxim)
    maxim = it.next();
```

seria incorrecta, perquè estem avançant dues vegades l'iterador en comptes d'una, i l'element que ens retorna a la primera i segona crida del **next** és diferent. Per això he utilitzat una variable **n** auxiliar per emmagatzemar el valor de l'element, i així només cridar a **next** una vegada.

A més, el podem generalitzar encara una mica més. En el següent exemple hem eliminat les crides a **size** i **get**, i ho fem tot a través de l'iterador. Per tant, no cal que suposem que ens passen una llista, sinó que podem utilitzar-lo sobre qualsevol col·lecció, com per exemple, un conjunt (**Set**):

```
public static int max(Collection<Integer> iterable) {
    Iterator<Integer> it;
    int maxim = 0;
    int n;

    it = iterable.iterator();
    if (it.hasNext())
        maxim = it.next();
    else
        throw new NoSuchElementException();

    while (it.hasNext()) {
        n = it.next();
        if (n > maxim)
            maxim = n;
    }

    return maxim;
}
```

47.8. Exemple d'implementació d'un iterador

E següent exemple mostra la implementació de dos iteradors, un per una llista enllaçada, i un per un array.

La implementació s'ha simplificat el màxim possible, deixant només els mètodes imprescindibles de cadascuna de les implementacions de llista.

```
public interface ElMeuIterator {  
    boolean hasNext();  
    int next();  
}
```

```
public interface LaMevaList {  
    void add(int n);  
    ElMeuIterator iterator();  
}
```

```
public class IntegerArrayList implements LaMevaList {  
    private int[] array = new int[100];  
    private int size;  
  
    @Override  
    public void add(int n) {  
        array[size]=n;  
        size++;  
    }  
  
    @Override  
    public ElMeuIterator iterator() {  
        return new IntegerArrayListIterator(this);  
    }  
  
    private class IntegerArrayListIterator implements ElMeuIterator {  
        private int pos;  
  
        private IntegerArrayList list;  
  
        public IntegerArrayListIterator(IntegerArrayList list) {  
            this.list = list;  
        }  
    }  
}
```

```
@Override  
public boolean hasNext() {  
    return pos < list.size;  
}  
  
@Override  
public int next() {  
    int n = list.array[pos];  
    pos++;  
    return n;  
}  
}
```

```
public class IntegerLinkedList implements LaMevaList {  
    private Node top;  
    private Node bottom;  
  
    @Override  
    public void add(int n) {  
        Node newNode = new Node(n);  
        if (top==null) {  
            top=bottom=newNode;  
        } else {  
            newNode.prev = bottom.prev;  
            bottom.next = newNode;  
            bottom = newNode;  
        }  
    }  
  
    @Override  
    public ElMeuIterator iterator() {  
        return new IntegerLinkedListIterator(top);  
    }  
  
    private class Node {  
        int n;  
        Node next;  
        Node prev;  
  
        public Node(int n) {  
            this.n=n;  
        }  
    }  
}
```

```
private class IntegerLinkedListIterator implements ElMeuIterator {  
    private Node pos;  
  
    public IntegerLinkedListIterator(Node top) {  
        pos = top;  
    }  
  
    @Override  
    public boolean hasNext() {  
        return pos != null;  
    }  
  
    @Override  
    public int next() {  
        int n = pos.n;  
        pos=pos.next;  
        return n;  
    }  
}
```

```
public class Client {  
  
    public static void main(String[] args) {  
        //LaMevaList ll = new IntegerLinkedList();  
        LaMevaList ll = new IntegerArrayList();  
        ll.add(2);  
        ll.add(4);  
        ll.add(6);  
        ll.add(8);  
        ll.add(10);  
        ll.add(12);  
        ll.add(14);  
        ll.add(16);  
  
        met(ll);  
    }  
  
    public static void met(LaMevaList ll) {  
        ElMeuIterator it = ll.iterator();  
        while (it.hasNext()) {  
            System.out.println(it.next());  
        }  
    }  
}
```

```
    }
}
}
```

Finalment, aquí podeu consultar un altre exemple d'ús d'iteradors. Noteu com l'ús de les llistes i els iteradors no depèn de la implementació concreta que estem utilitzant:

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

public class Main {

    public static void main(String[] args) {
        String[] cadenes = {"a", "b", "c", "d", "b", "d", "a", "c", "b", "c"};
        List<String> llista1 = new ArrayList<>(Arrays.asList(cadenes));
        List<String> llista2 = new LinkedList<>(Arrays.asList(cadenes));

        eliminaElement(llista1, "b");
        eliminaElement(llista2, "d");
        mostraLlista(llista1);
        System.out.println();
        mostraLlista(llista2);
    }

    public static void eliminaElement(List<String> llista, String
aEliminar) {
        Iterator<String> it = llista.iterator();

        while (it.hasNext()) {
            if (it.next().equals(aEliminar))
                it.remove();
        }
    }

    public static void mostraLlista(List<String> llista) {
        for (String s : llista) {
            System.out.println(s);
        }
    }
}
```

```
}
```

47.8.1. Els iteradors de llistes: **ListIterator**

Fixeu-vos que podem utilitzar iteradors també sobre conjunts. Encara que en un conjunt no es pugui garantir l'ordre dels seus elements, res impedeix que els puguem recórrer tots. En el cas de les llistes, com que a més tenen ordre, existeix un tipus especialitzat d'iterador, el **ListIterator**, i el mètode **listIterator()** de la interfície **List** per obtenir-ne un exemplar. Amb un **ListIterator**, a més de recórrer els elements des del primer fins a l'últim, ho podem fer al revés, mouent l'iterador en el sentit que ens interessi i inicialitzant-lo a la posició que vulguem.

Un **ListIterator** té tots els mètodes d'un **Iterator** (la interfície **ListIterator** estén la **Iterator**), i afegeix els mètodes **previous()** i **hasPrevious()** anàlegs a **next()** i **hasNext()**, però però recórrer la llista en sentit contrari. A més, el mètode **listIterator** pot rebre un enter, tal que la primer crida a **next** serà l'element que ocupa la posició indicada per aquest enter.

Ens hem d'imaginar que un iterador d'aquest tipus es troba entremig de dos elements, que una crida a **next** retornarà el de la dreta, i una crida a **previous** retornarà el de l'esquerra:



El següent codi recorreria una llista començant per l'últim element i acabant pel primer:

```
Integer i;
ListIterator<Integer> it = l.listIterator(l.size());
while (it.hasPrevious()) {
    i = it.previous();
    // fer alguna cosa amb i
}
```

47.8.2. Inserir i extreure elements a través d'un iterador

A més dels mètodes que hem vist, la interfície **ListIterator** també incorpora mètodes per afegir i extreure elements a la llista a la posició on es troba l'iterador, així com per reemplaçar l'element per un altre.

Quan treballem amb iteradors és important recordar que només podem modificar la llista a través dels mètodes de l'iterador. L'iterador emmagatzema la informació necessària sobre l'estat de la llista per tal de poder retornar els elements anterior i següent de forma consistent. Si modifiquem la llista sense que l'iterador ho sàpiga, cridant per exemple el mètode **add** o **remove** de la llista directament, la informació que té l'iterador ja no serà coherent amb la realitat de la llista, i ens trobarem amb problemes quan intentem seguir amb el recorregut.

Com a corol·lari de l'anterior, cal recordar que els iteradors s'han d'inicialitzar just en el moment d'utilitzar-los, per evitar que hi hagi modificacions a la llista entre el moment en què s'inicialitza i el moment en què s'utilitza. No hi ha problema amb declarar la variable al principi, però la crida a **iterator()** o **listIterator()** s'ha de fer quan realment volem utilitzar l'iterador.

47.8.3. Ús d'un iterador sense l'iterador: enhanced for

Es coneix amb el nom de **enhanced for** una variant del bucle **for** que permet recórrer per tots els elements d'una col·lecció. Aquest és el **for** per defecte que utilitzen molts llenguatges de scripting, com el Bash o el Python.

En Java, aquest tipus de bucle s'escriu així:

```
for (Integer i : c) {  
    System.out.println(i);  
}
```

on **c** és una col·lecció qualsevol que conté enters.

Aquest codi és equivalent a:

```
Iterator<Integer> it = c.iterator();  
while (it.hasNext())  
    System.out.println(it.next());
```

L'**enhanced for** es va afegir al llenguatge Java a la seva versió 5, precisament per simplificar l'ús dels iteradors. És clar que no és aplicable a totes les situacions, i que treballar amb els iteradors segueix essent necessari, per exemple, si volem afegir elements durant el recorregut, o alternar l'ordre del recorregut, però per a les situacions més comunes, és una alternativa elegant.

Anem a veure'n algunes possibilitats en el següent exemple.

Imaginem que tenim dues llistes, una amb els pals d'una baralla de cartes, i l'altre amb els nombres possibles de les cartes. Volem crear una tercera llista amb totes les cartes de la baralla, combinant cada pal possible amb cada número possible.

Podríem intentar-ho així:

```
List<Pal> pals = ...;
List<Numero> numeros = ...;
List<Carta> baralla = new ArrayList<Carta>();

// ERROR: llença NoSuchElementException!
for (Iterator<Pal> i = pals.iterator(); i.hasNext(); )
    for (Iterator<Numero> j = numeros.iterator(); j.hasNext(); )
        baralla.add(new Carta(i.next(), j.next()));
```

Quin és l'error en aquest codi?

Doncs que el mètode **next** per l'iterador **i** es crida moltes vegades dins del bucle intern. Cada vegada ens retorna un pal diferent, i no s'està comprovant quan s'acaben amb el **hasNext**.

Podríem arreglar-ho així:

```
for (Iterator<Pal> i = pals.iterator(); i.hasNext(); ) {
    Pal pal = i.next();
    for (Iterator<Numero> j = numeros.iterator(); j.hasNext(); )
        baralla.add(new Carta(pal, j.next()));
}
```

Ara ens hem assegurat que el **next** d'**i** es crida un sol cop per iteració, però el codi és una miqueta complicat de llegir. Amb un bucle **enhanced for** la solució queda sorprendentment simple:

```
for (Pal pal : pals)
    for (Numero num : numeros)
        baralla.add(new Carta(pal, num));
```

A més d'això, és important destacar que els bucles **enhanced for** també es poden utilitzar amb els vectors estàtics de Java:

```
// Retorna la suma dels elements d'a:
int suma(int[] a) {
    int resultat = 0;
    for (int i : a)
        resultat += i;
    return resultat;
}
```

47.9. Algorismes: les classes Arrays i Collections

Recuperem el mètode **max** que hem creat a la secció sobre els iteradors. A Java tenim mètodes d'aquest tipus que encara són més generals. En comptes de tenir un mètode **max** per **Integer**, i haver-ne de fer un de diferent per a cada tipus de dada, ens ho han fet de manera que els podem utilitzar per a qualsevol cosa que es pugui comparar amb elements del mateix tipus. Els tenim a la classe **Collections**:

```
public static <T extends Object & Comparable<? super T>> T
max(Collection<? extends T> coll)
```

Fixeu-vos quina declaració tan complexa!

Independentment que no entenguem aquesta declaració, l'ús del mètode queda clar: li passem una col·lecció de qualsevol tipus, que contingui elements de qualsevol tipus, i ell ens retorna l'element més gran que hi ha a la col·lecció. L'única restricció és que els elements de la col·lecció es puguin comparar entre ells, ja que si no, no té sentit parlar del seu màxim. Així, ho podrem fer servir per col·leccions que continguin **Integer**, **Double**, **String**... i qualsevol altra classe que implementi la interfície **Comparable**. Aquesta interfície s'utilitza per indicar que els elements d'una certa classe es poden comparar, i la veurem amb detall al tema de programació orientada a objectes.

La classe **Collections** conté una sèrie de mètodes estàtics útils per treballar amb col·leccions. Per exemple, tenim el mètode **sort** per ordenar una col·lecció, el mètode **swap** per intercanviar dos elements de la col·lecció, el **shuffle** per barrejar la col·lecció i el **copy** per copiar una col·lecció, entre d'altres.

La classe **Arrays** és similar, però conté mètodes útils per treballar amb vectors estàtics, algun dels quals ja ha anat apareixent al llarg dels exemples del curs.

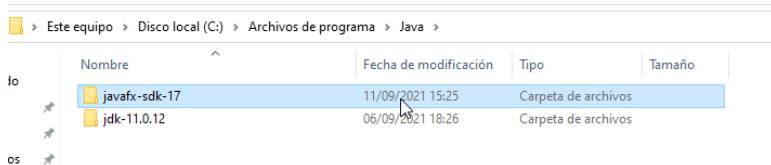
47.10. Configuració de JavaFX i Apache Netbeans amb Ant

A partir de Java8 JavaFX no es distribueix dins del JavaJDK, es distribueix com un modul a part.

47.10.1. Baixar i instal·lar la biblioteca

En primer lloc, cal obtenir la biblioteca JavaFX. La podeu trobar a Gluon JavaFX. Baixeu el SDK de la última versió LTS corresponent al vostre sistema operatiu.

1. Haureu de descomprimir el fitxer en una carpeta, en una ubicació coneguda de la vostra màquina, per exemple:

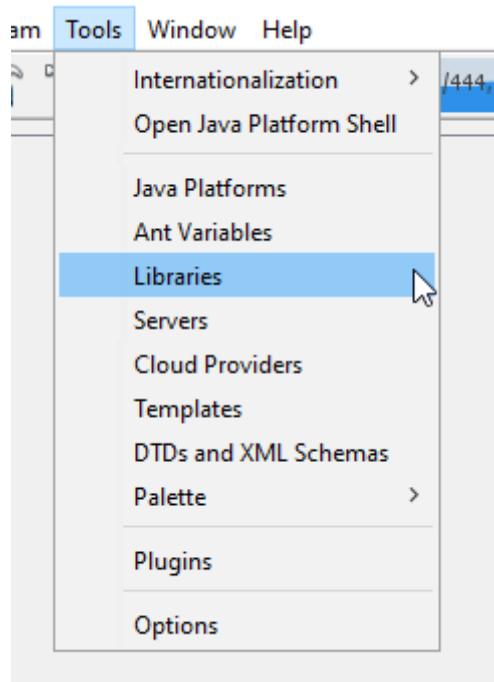


Un cop descomprimida la biblioteca, tot el que heu de fer és afegir-la a qualsevol projecte Java existent o nou en el qual vulgueu codificar amb JavaFX. Aquesta part pot ser una mica molesta ja que cal fer-ho per a cada projecte JavaFX.

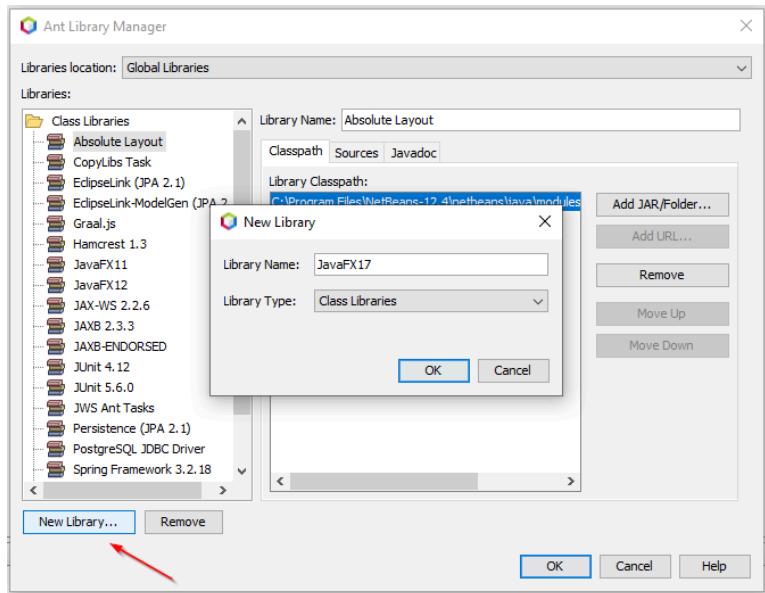
47.10.2. Afegir la biblioteca JavaFX globalment

En primer lloc, hem de crear una nova biblioteca global per a la biblioteca JavaFX, això afegirà un element de biblioteca per a la biblioteca JavaFX que després podeu afegir a cada projecte Java en el qual voleu utilitzar JavaFX.

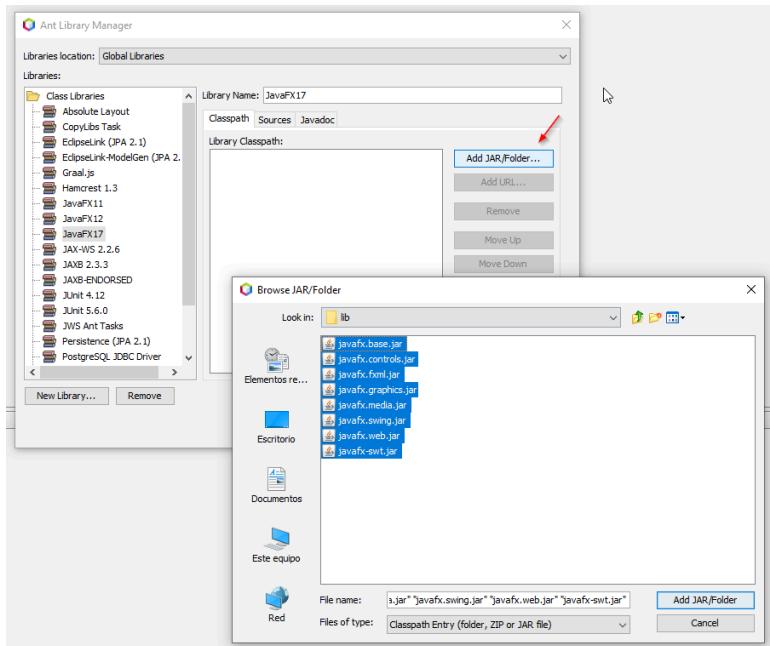
1. Al menú **Tools**, seleccioneu **Libraries**.



2. Veureu una llista de biblioteques existents a l'esquerra i una àrea al mig on podeu afegir altres biblioteques. A sota de la llista de biblioteques de l'esquerra, feu clic al botó **New Library**. Això obrirà un diàleg on podeu introduir la informació de la biblioteca JavaFX.



3. Poseu el nom a la biblioteca, en l'exemple JavaFX17 i feu clic al botó **OK** per tornar a la pantalla administrador de biblioteques.
4. La biblioteca JavaFX hauria d'aparèixer a la llista Biblioteques de l'esquerra i actualment seleccionada, de manera que la seva informació aparegui a l'àrea principal de la pantalla.
5. A la part dreta de la pantalla, feu clic al botó **Add JAR/Folder** per afegir els fitxers que componen la biblioteca JavaFX.
6. Navegueu fins a la ubicació on heu descomprimit el fitxer zip JavaFX, entreu a la carpeta **lib**, i seleccioneu tots els fitxers amb l'estensió **.jar**. No seleccioneu el fitxer **src.zip** o les propietats (si n'hi ha), **seleccioneu només** els fitxers **JAR**.

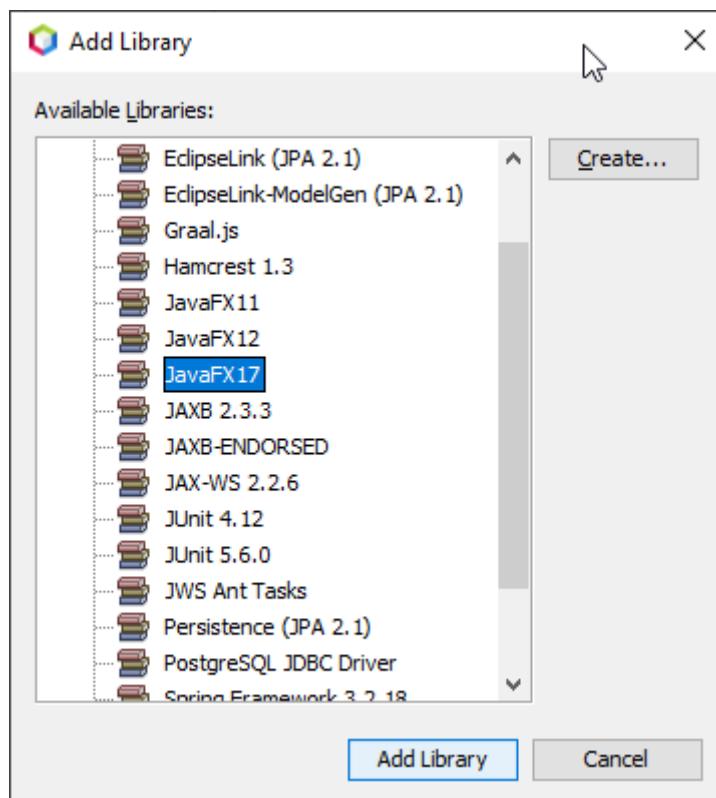
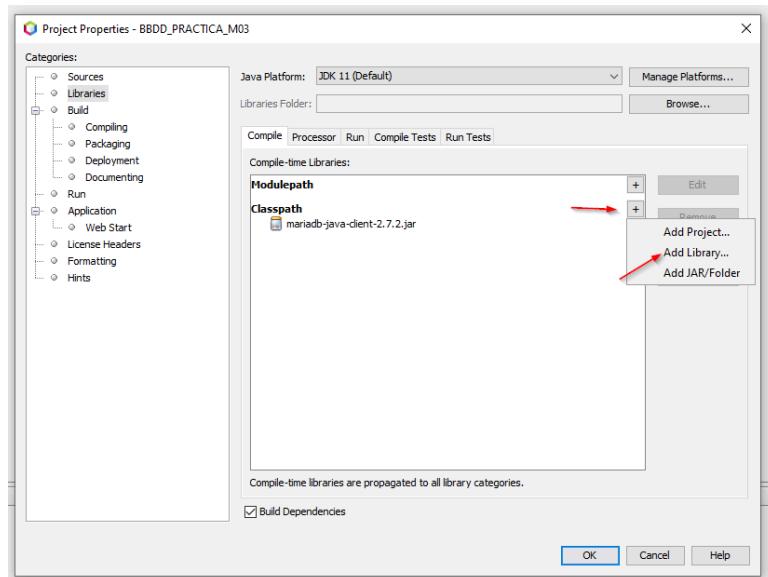


7. Reinicie netbeans

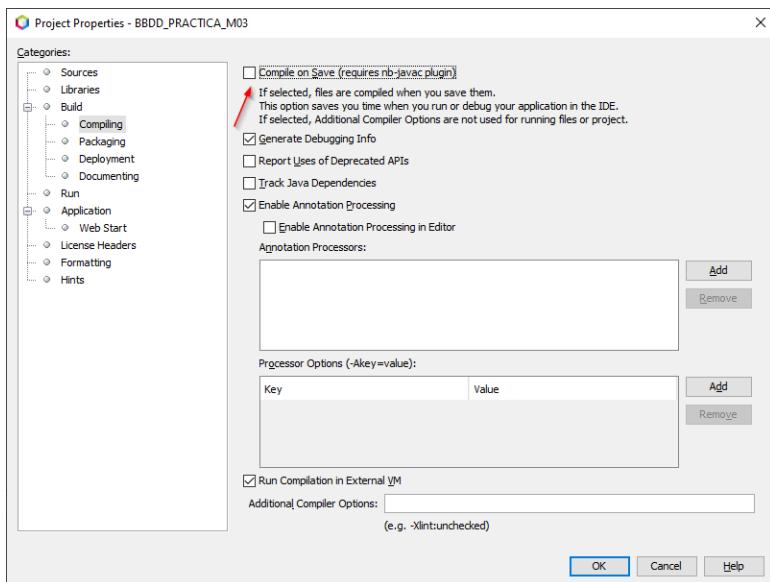
47.10.3. Afegir JavaFX a un projecte concret

1. Crear un projecte nou de la manera habitual **no creeu el projecte JavaFX**, no funciona.
2. Feu clic amb el botó dret al nom del nou projecte seleccioneu **Properties**. Al diàleg Propietats, feu clic al node **Libraries** del costat esquerre. Afegiu ñla llibreria JavaFX creada als passos anteriors.

Afegir JavaFX a un projecte concret



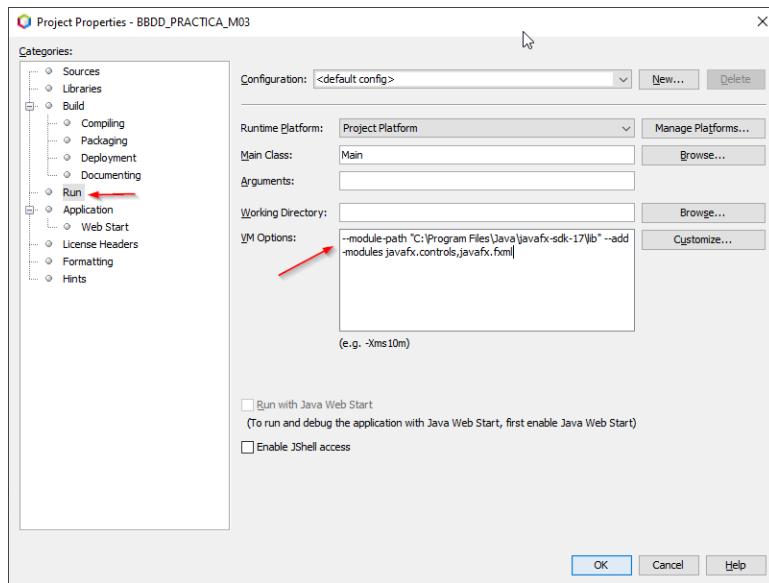
3. Hi ha dues propietats de projecte més que haureu de canviar perquè JavaFX funcioni, desactivar **properties > Build > Compiling > Compile on Save**.



4. Afegiu la ruta del mòdul per a dos dels fitxers jar biblioteca JavaFX alhora de compilar. Aneu a les propietats del projecte > **Run** i al camp **VM Options**, copieu i enganxeu el següent (adequant-ho a les vostres rutes):

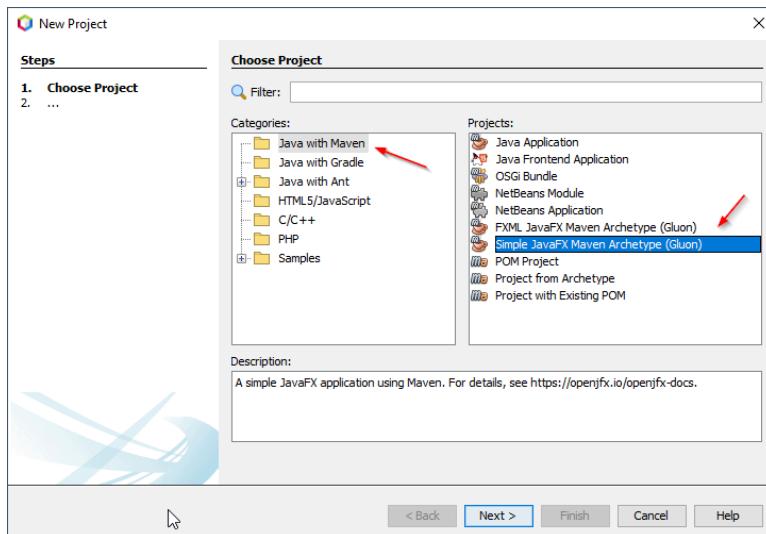
```
--module-path "C:\Program Files\Java\javafx-sdk-17\lib" --add-modules javafx.controls,javafx.fxml
```

Configuració de JavaFX i Apache Netbeans amb Maven



47.11. Configuració de JavaFX i Apache Netbeans amb Maven

Per treballar en JavaFX amb un projecte en maven n'hi ha prou en seleccionar les següents opcions alhora de crear el projecte.



Convencions del groupId, artifactId i versió

groupId

Identifica el projecte de manera unívoca. El **groupId** hauria de seguir les mateixes regles que per anomenar paquets d eJava, és a dir, **c^omencar per un nom de domini prpoi invertit**. Per exemple:

```
cat.copernic
```

artifactId

És el nom de **jar** sense la versió. ÉS possible posar qualsevol nom sempre i quant estigui **tot en minúscules i no tingui caràcters extranets**

version

Si s'ha de distribuir el paquet es pot posar qualsevol numero de versió amb el format x.x.x. Per exemple:

```
1.0.1
```

Si la versió no s'ha distribuït maven utilitza la notació 1.0.SNAPSHOT.

48

JavaFX - Introducció

48.1. Introducció a les interfícies gràfiques en Java

Des que el Java va aparèixer a mitjan dels noranta, la forma d'implementar interfícies gràfiques ha anat evolucionant. Per tal de mantenir la compatibilitat amb versions anteriors, les classes originals no s'han eliminat de les API, de manera que ara mateix conviuen diversos sistemes de finestres:

AWT (Abstract Window Toolkit)

En les primeres versions de Java, les classes que gestionaven la GUI (Graphical User Interface) estaven incorporades en una llibreria anomenada **AWT** (Abstract Window ToolKit)

AWT estava dissenyat per construir GUI simples, no era enterament independent de plataforma i tenia una quantitat important d'errors relacionats en general amb la portabilitat.

Swing

Fins fa poc era el conjunt d'eines recomanat per Oracle per a implementar interfícies gràfiques en Java. A diferència de l'AWT, les classes de Swing estan implementades en Java i són, per tant, independents del sistema operatiu en què s'executen.

La biblioteca **Swing** va reemplaçar **AWT**, Swing era més versàtil, més robusta i més flexible que AWT. Estava dissenyada per desenvolupar **interfícies d'usuari d'escriptori**

La biblioteca Swing **ja no està en desenvolupament** i només es mantenen les últimes versions per un a qüestió de compatibilitat.

La biblioteca que es considera estàndard pel desenvolupament d'interfícies d'usuari en les últimes versions de Java s'anomena **JavaFX**.

JavaFX

A diferència de Swing JavaFX està dissenyada per la creació de GUIs en aplicacions **RIA (Rich Internet Application)**, és a dir, aplicacions web amb funcionalitats que normalment estan associades a aplicacions d'escriptori.

Una aplicació JavaFX pot executar-se sobre un escriptori o sobre un navegador web.

SWT (Standard Widget Toolkit)

Aquest és un conjunt d'eines per a construir interfícies gràfiques que no està inclòs a la JRE/JDK, però que mereix especial atenció degut a què hi ha projectes prou importants que l'utilitzen, entre d'ells, Eclipse (que són els que mantenen SWT actualment) i diversos projectes d'IBM (que en són els creadors originals).

El projecte **RAP (Remote Application Platform)** uneix SWT amb altres tecnologies per al desenvolupament d'aplicacions en xarxa.

48.2. Estructura bàsica d'un programa de JavaFX

Hello World en JavaFX.

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.stage.Stage;

public class HelloWorldJavaFX extends Application { ①

    @Override
    public void start(Stage primaryStage) { ②
        Button rootNode = new Button("Hello World"); ③
        Scene scene = new Scene(rootNode, 200, 250); ④
        primaryStage.setTitle("Hello World");
        primaryStage.setScene(scene); ⑤
        primaryStage.show(); ⑥
    }
}
```

```

    }

    public static void main(String[] args) {
        Application.launch(args); ⑦
    }
}

```

- ❶ Les aplicacions de JavaFX deriven de la classe **javafx.application.Application**, classe que proporciona les funcions associades al cicle de vida de l'aplicació.
- ❷ El mètode **launch** passa el control del fil principal al motor de JavaFX i passa els arguments passats a l'aplicació per línia de comandes a JavaFX. Aquest crida el mètode **start en un nou fil d'execució**, un fil d'execució propi de JavaFX que gestiona la UI.
- ❸ La UI en JavaFX es disposa en una estructura de graf començant per un node arrel. A l'exemple és un botó però normalment serà algun tipus de node contenidor.
- ❹ En primer lloc tenim el node **Stage** (escenari), aquest node representa la finestra on es dibuixarà la UI.
- ❺ Just a sota podem tenir un node **Scene** (escena), que representa el panell on es posaran els diferents elements de la interfície d'usuari. Finalment a sota dels nodes **Scene** hi pengen **Nodes** que representen els diferents controls que conformaran la UI, botons, desplegables, etiquetes, etc...
- ❻ Un cop tot preparat es mostra el *stage*.



Podríem tenir una segona **Scene** i associar-la com activa al **Stage** en qualsevol moment.



Alguns d'aquests **Nodes** poden ser pares d'altres nodes.

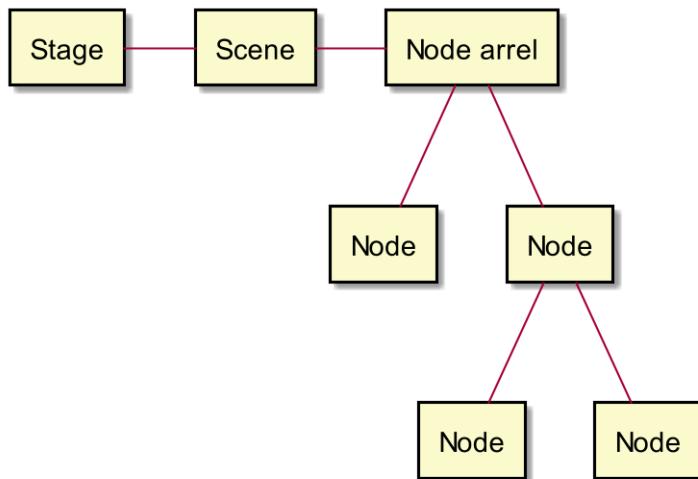
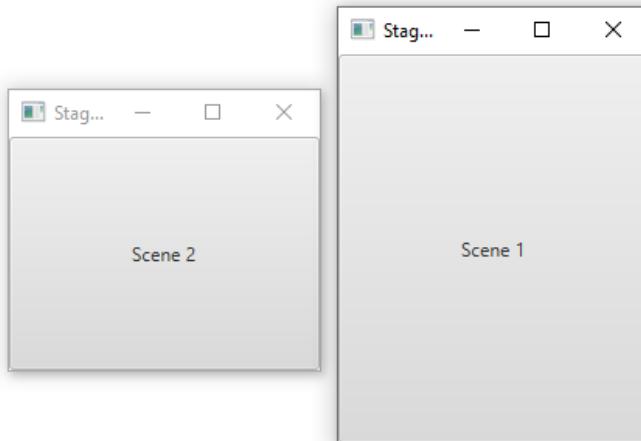


Figura 48.1. Graf amb una escena de JavaFX

48.3. Aplicacions amb més d'un Stage

És possible crear una aplicació amb més d'un **Stage**, en aquest cas cada **Stage** té la seva pròpia escena amb la seva estructura de nodes.

El següent exemple mostra com fer-ho:



Una aplicació amb dues escenes.

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.stage.Stage;

public class MultiStageDemo extends Application {

    @Override
    public void start(Stage primaryStage) {

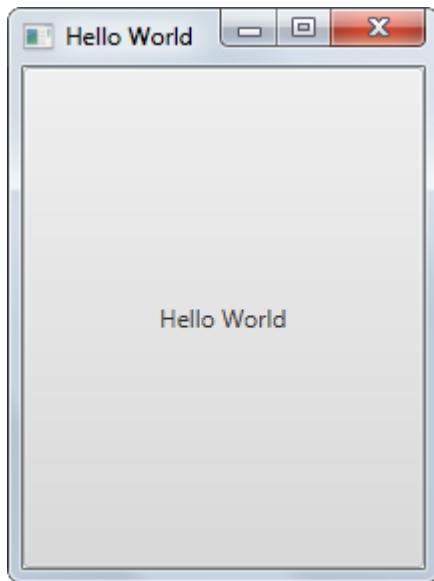
        Scene scene = new Scene(new Button("Scene 1"), 200, 250);
        primaryStage.setTitle("Stage 1");
        primaryStage.setScene(scene);
        primaryStage.show();

        Stage stage = new Stage();
        stage.setTitle("Stage 2");
        stage.setScene(new Scene(new Button("Scene 2"), 200, 150));
        stage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

48.4. Panells, Controls i Formes

Si ens fixem en el resultat de la exemple anterior veiem que el **botó ocupa tota la finestra independentment de la mida d'aquesta**.



Per sol·lucionar-ho tenim dues opcions:

1. Configurar manualment les propietats del botó (mida i posició).



Aquest enfoc **no** és el recomanat ja que la resulta en UI no portables, interfícies d'usuari que només es veuen correctament en pantalles amb la mateixa resolució i proporcions que la pantalla amb la que s'han desenvolupat fruit de treballar en coordenades absolutes.

2. Utilitzar algun tipus de **classe contenidor**, classes derivades de **javafx.scene.layout.Pane**, que permeten organitzar automàticament els elements de la UI independentment de les mides del contenidor.

La classe **Shape** fa referència a una forma elemental, una línia, un cercle, etc...

La classe **ImageView** és un node que s'utilitza per mostrar imatges carregades amb la classe **Image**.

La classe **Parent** és la classe base per tots els nodes que poden tenir fills.

La classe **Region** és la classe base per a tots els controls visuals derivats de **Node** i tots els contenidors de composició (**layout**). Les regions tenen, vora i fons i tenen forma de rectangle.

La classe **Control** fa referència a un control en una UI, un botó, una etiqueta, una àrea de text, etc...

A continuació es mostra un diagrama amb la relació entre les classes **Node**, **Control**, **Pane** i algunes classes més de javafx.

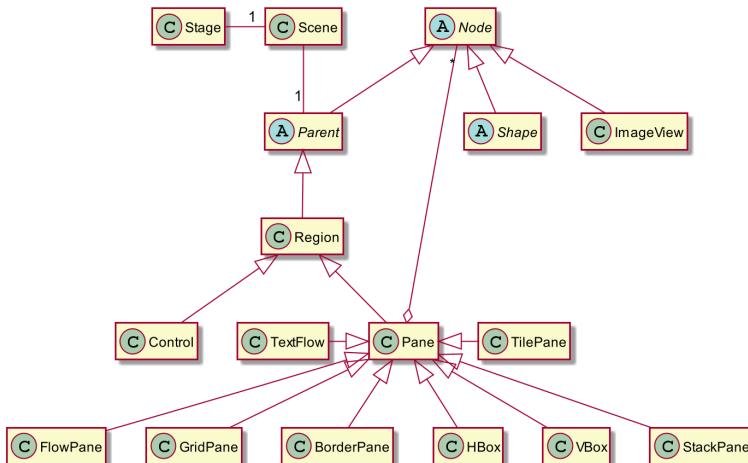


Figura 48.2. Relació entre les classes **Node**, **Control** i **Pane**



Fixeu-vos que segons el diagrama anterior una **Scene** pot contenir un **Control** o un **Pane** però no pot contenir directament una **Shape** o un **imageView**.

No obstant, un **Pane** pot contenir objectes de qualsevol classe derivada de **Node**.

48.5. Com s'utilitza un **Pane** amb un **Control**

Exemple elemental de la classe **Control**.

```

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.stage.Stage;
  
```

```

import javafx.scene.layout.StackPane;

public class PaneDemo extends Application {

    @Override
    public void start(Stage primaryStage) {

        StackPane rootNode = new StackPane(); ①
        rootNode.getChildren().add(new Button("Hello World")); ②
        Scene scene = new Scene(rootNode, 200, 50);
        primaryStage.setTitle("Pane Demo");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}

```

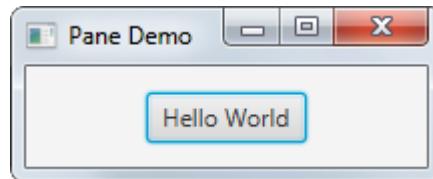


Figura 48.3. Exemple elemental de la classe Control

- ➊ Hem modificat el programa HelloWorld afegint un **Node** de tipus **Pane** com a **node arrel**. En particular un **StackPane**, aquest **Pane** posa els nodes al seu centre i els va superposant en ordre de col·locació. Els nodes afegits a un **StackPane** respecten la seva **preferredSize**, la seva mida per defecte.
- ➋ Afegim el botó com a fill del **StackPane**.



El mètode **getChildren()** de **Node** retorna una instància de **javafx.collections.ObservableList**. **ObservableList** és similar en funcionament a una **ArrayList** amb la particularitat que implementa el patró **Observer**.

48.6. La classe Pane

La classe **Pane** s'utilitza quan es requereix un posicionament absolut dels fills.

No s'acostuma a utilitzar quan cal treballar amb UI de controls ja que la disposició d'aquests no depèn ni de la mida de la pantalla ni de la resolució cosa que genera codi poc portable.

48.7. Com s'utilitza un **Pane** amb una **Shape**

En lloc de posar objectes de la classe **Control** a dins d'un **Pane** podem dibuixar-hi directament.

Per fer-ho podem prendre directament un objecte de la classe **Pane** i utilitzar-lo com a superfície de dibuix. Per dibuixar-hi anirem afegint a l'arbre de nodes objectes derivats de la classe **Shape** que representen les diferents primitives de dibuix de JavaFX.

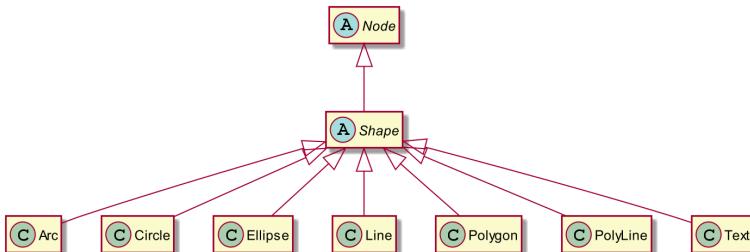


Figura 48.4. Part de la jerarquia de classes de **Shape**

Vegem-ho amb un exemple senzill:

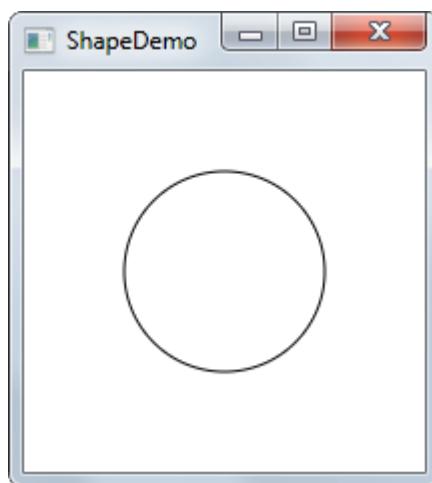


Figura 48.5. Exemple elemental de la classe **Shape**

Exemple elemental de la classe Shape.

```

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;

public class ShapeDemo extends Application {

    @Override
    public void start(Stage primaryStage) {
        Circle circle = new Circle(); 1
        circle.setCenterX(100); 2
        circle.setCenterY(100); 3
        circle.setRadius(50); 4
        circle.setStroke(Color.BLACK); 5
        circle.setFill(Color.WHITE); 6

        Pane rootNode = new Pane();
        rootNode.getChildren().add(circle);

        Scene scene = new Scene(rootNode, 200, 200);
        primaryStage.setTitle("ShapeDemo");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}

```

- ⑥ Creem un **Circle**, que deriva de **Shape** i el configurem.**

Coses importants:

1. Per defecte l'origen de coordenades es troba a la part superior esquerra del **Pane** i les coordenades són positives tant en l'eix x com en l'eix y.
2. Totes les mesures en JavaFX es donen en *pixels*.

3. Per especificar un "no color" es pot utilitzar *null* en els mètodes on es demani un color.

48.8. Classe Color

La classe **Color** s'utilitza per crear colors.

C	Color
□	red: double
□	green: double
□	blue: double
□	opacity: double
●	Color(r: double, g: double, b: double, opacity: double)
●	brighter(): Color
●	darker(): Color
●	<u>color(r: double, g: double, b: double): Color</u>
●	<u>color(r: double, g: double, b: double, opacity: double): Color</u>
●	<u>rgb(r: int, g: int, b: int): Color</u>
●	<u>rgb(r: int, g: int, b: int, opacity: double): Color</u>

Figura 48.6. javafx.scene.paint.Color

48.9. Classe Font

La classe **Font** descriu el nom de la font, la mida i les possibles decoracions, cursiva, negreta, etc...

C	Font
□	size: double
□	name: String
□	family: String
●	Font(size: double)
●	Font(name: String, size: double)
●	<u>font(name: String, size: double)</u>
●	<u>font(name: String, w : FontWeight, size: double)</u>
●	<u>font(name: String, w : FontWeight, p: FontPosture, size: double)</u>
●	<u>getFamilies(): List<String></u>
●	<u>getFontNames(): List<String></u>

Figura 48.7. javafx.scene.text.Font

getFamilies(): List<String>

Retorna una llista dels noms de les famílies de les diferents fonts.

getFontNames(): List<String>

Retorna una llista dels noms de les diferents fonts.

Podem crear un objecte **Font** de la següent manera:

```
Font font1 = new Font("SansSerif", 16);
Font font2 = Font.font("Times New Roman", FontWeight.BOLD,
FontPosture.ITALIC, 12);
```



```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.*;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.text.*;
import javafx.scene.control.*;
import javafx.stage.Stage;

public class FontDemo extends Application {

    @Override
    public void start(Stage primaryStage) {
```

```

Pane pane = new StackPane();

Circle circle = new Circle();
circle.setRadius(50);
circle.setStroke(Color.BLACK);
circle.setFill(new Color(0.5, 0.5, 0.5, 0.1));
pane.getChildren().add(circle);

Label label = new Label("Hola!!!");
label.setFont(Font.font("Times New Roman", FontWeight.BOLD,
FontPosture.ITALIC, 20));
pane.getChildren().add(label);

Scene scene = new Scene(pane);
primaryStage.setTitle("Font Demo");
primaryStage.setScene(scene);
primaryStage.show();
}
}

```

48.10. Aplicar fulls d'estil CSS

JavaFX permet aplicar fulls d'estils CSS a les escenes i als diferents nodes d'una escena.

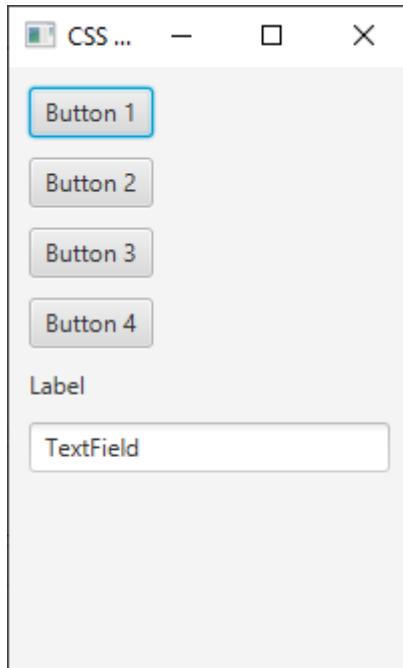
El llenguatge que s'utilitza per elaborar els diferents estils **és similar al que s'utilitza per les pàgines web però no té els mateixos atributs**.



D'entrada JavaFX ja utilitza un full d'estils **predeterminat** que es troba al fitxer **jfxrt.jar**.

És important tenir-ho en compte quan es defineixen noves classes CSS ja que si el nom de les classes coincideix amb alguna de les classes predefinides estarem sobreescrivint els estils i s'aplicaran directament, si no, estarem definint una classe nova i l'haurem d'aplicar manualment a l'element, no s'aplicarà directament.

Per veure'n les possibilitats partim de la següent escena:



```
package javafxcsstest;

import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class JavaFXcssTest extends Application {

    @Override
    public void start(Stage primaryStage) {

        VBox root = new VBox();
        root.setPadding(new Insets(10, 10, 10, 10));
        root.setSpacing(10);

        Button btn1 = new Button("Button 1");
        Button btn2 = new Button("Button 2");
        Label label = new Label("Label");
        TextField textField = new TextField("TextField");

        root.getChildren().addAll(btn1, btn2, label, textField);
        primaryStage.setScene(new Scene(root));
        primaryStage.show();
    }
}
```

```
        Button btn3 = new Button("Button 3");
        Button btn4 = new Button("Button 4");
        Label lbl = new Label("Label");
        TextField txt = new TextField("TextField");

        root.getChildren().addAll(btn1, btn2, btn3, btn4, lbl, txt);

        Scene scene = new Scene(root, 200, 300);

        primaryStage.setTitle("CSS Test");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Per aplicar estils a un node podem treballar de dues maneres:

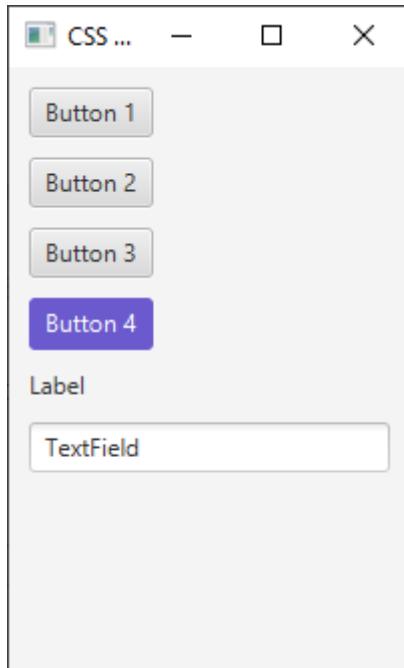
- Aplicar els estils directament amb el mètode **setStyle()** del propi node.
- Crear un full d'estils **css**, i aplicar-lo directament a la escena o a un node.

48.10.1. Aplicar un estil directament

Podem aplicar un estil directament de la següent manera:

```
Button btn4 = new Button("Button 4");
btn4.setStyle("-fx-background-color: slateblue; -fx-text-fill: white;");
```

Cos que ens donaria:



48.10.2. Crear un full d'estils i aplicar-lo a la escena

Podem crear un full d'estils i incorporar-lo al projecte:

style.css.

```
.root{ ❶
    -fx-font-size: 16pt;
    -fx-font-family: "Courier New";
    -fx-base: rgb(132, 145, 47);
    -fx-background: rgb(225, 228, 203);
}

.button{ ❷
    -fx-text-fill: rgb(49, 89, 23);
    -fx-border-color: rgb(49, 89, 23);
    -fx-border-radius: 5;
    -fx-padding: 3 6 6 6;
}

.button2{ ❸
    -fx-text-fill: #006464;
```

```

    -fx-background-color: #DFB951;
    -fx-border-radius: 20;
    -fx-background-radius: 20;
    -fx-padding: 5;
}

#font-button { ④
    -fx-font: bold italic 20pt "Arial";
    -fx-effect: dropshadow( one-pass-box , black , 8 , 0.0 , 2 , 0 );
}

```

- ① Sobreescrivim l'estil del node arrel de l'escena.
 - ② Apliquem un nou estil sobre tots els botons de l'escena.
 - ③ Creem una nova classe css que podrem aplicar a nodes específics utilitzant el mètode **getStyleClass().add()**.
 - ④ Creem un estil **id** que aplicarem sobre un node utilitzant el mètode **setId()**.
- Amb el full d'estils anterior podem modificar el codi Java de la següent manera:

```

package javafxcsstest;

import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class JavaFXcssTest extends Application {

    @Override
    public void start(Stage primaryStage) {

        VBox root = new VBox();
        root.setPadding(new Insets(10, 10, 10, 10));
        root.setSpacing(10);

        Button btn1 = new Button("Button 1");
        Button btn2 = new Button("Button 2");
        btn2.getStyleClass().add("button2"); ①
        Button btn3 = new Button("Button 3");
    }
}

```

```
btn3.getStyleClass().clear(); ②
btn3.setId("font-button");

Button btn4 = new Button("Button 4");
btn4.setStyle("-fx-background-color: slateblue; -fx-text-fill: white;");
Label lbl = new Label("Label");
TextField txt = new TextField("TextField");

root.getChildren().addAll(btn1, btn2, btn3, btn4, lbl, txt);

Scene scene = new Scene(root, 200, 300);
scene.getStylesheets().add("/styles/style.css"); ③

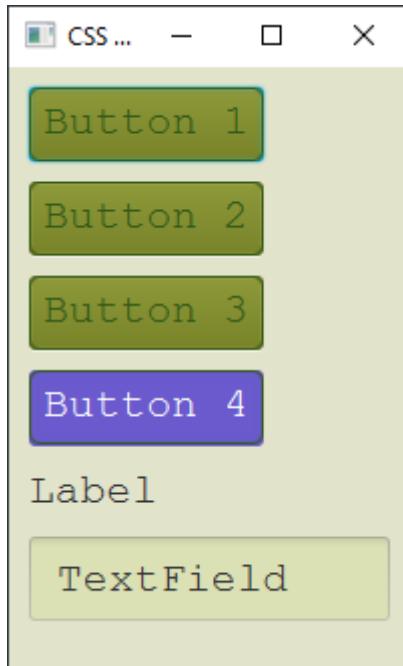
primaryStage.setTitle("CSS Test");
primaryStage.setScene(scene);
primaryStage.show();
}

public static void main(String[] args) {
    launch(args);
}
}
```

- ③ Afegim el full d'estils **style.css** als estils de l'escena.



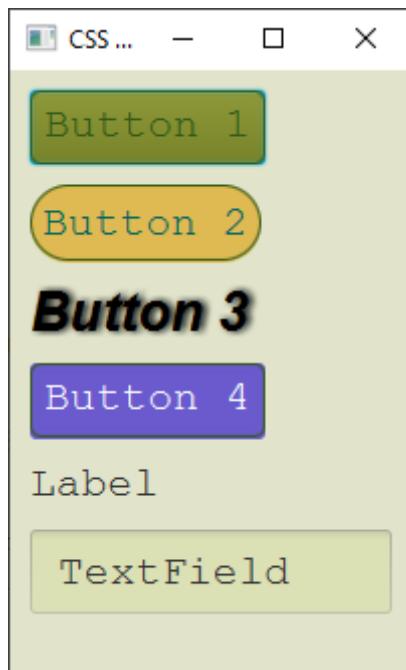
Si treballem amb **maven** cal deixar el fitxer amb els estils a la carpeta **main/resources/styles/**



- ① Afegim l'estil definit per la classe **button2** al botó.

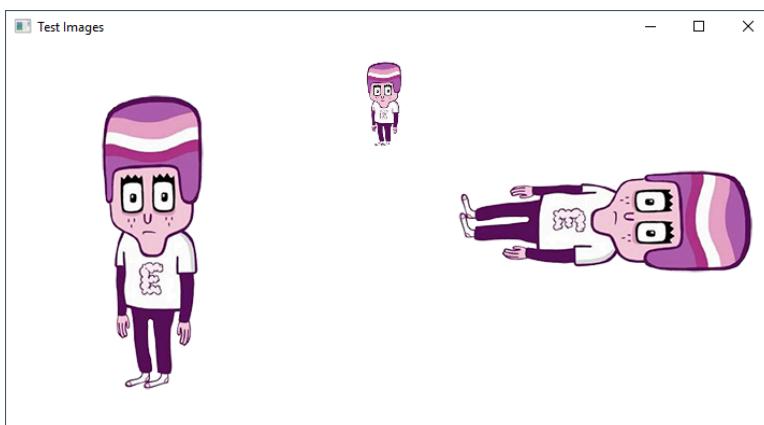


- ② Eliminem els estils prèviament definits al botó i li assignem el **id font-button** que té un estil propi associat.



48.11. Classe Image i ImageView

La classe **Image** representa una imatge en memòria, el control **ImageView** permet mostrar una **Image** per pantalla.



```
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.HBox;
import javafx.scene.layout.Pane;
import javafx.stage.Stage;

public class ImageTest extends Application {

    @Override
    public void start(Stage primaryStage) {

        Pane pane = new HBox(10); ①
        pane.setPadding(new Insets(5, 5, 5, 5));

        // En netbeans posem al imatge a src
        Image image = new Image("enjuto.png"); ②

        pane.getChildren().add(new ImageView(image)); ③

        ImageView imageView2 = new ImageView(image);
        imageView2.setFitHeight(100);
        imageView2.setFitWidth(100);
        pane.getChildren().add(imageView2);

        ImageView imageView3 = new ImageView(image);
        imageView3.setRotate(90);
        pane.getChildren().add(imageView3);

        Scene scene = new Scene(pane);
        primaryStage.setTitle("Test Images");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

- ① HBox és un **pane** que posa tots els nodes en una fila horitzontal.

- ② Creem una imatge.



Si treballem amb **maven** cal deixar la imatge a la carpeta **main/resources/**

- ③ L'afegeim a la ImageView.



48.12. Layouts

JavaFX proporciona diferents classes derivades de **Pane** per organitzar els **Nodes** en un contenidor.

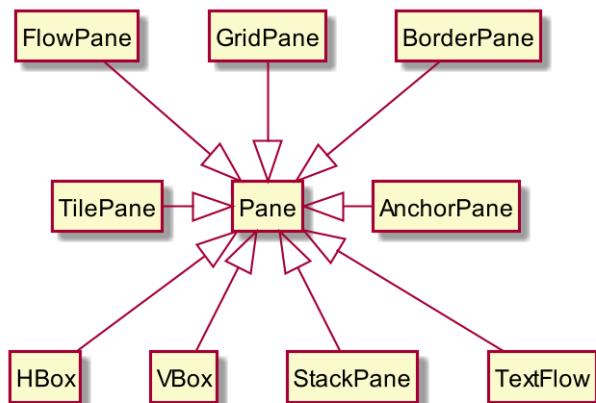


Figura 48.8. Layouts de JavaFX

Taula 48.1. Layouts de JavaFX

Classe	Descripció
Pane	Classe base dels diferents panells. Té el mètode getChildren() per accedir a la llista de Nodes del panell. S'utilitza quan es requereix un posicionament absolut dels nodes fill.
StackPane	Posa els nodes l'un damunt de l'altre al centre del panell.
FlowPane	Posa els nodes un darrera l'altre canviant si cal de fila o columna.

Classe	Descripció
TextFlow	TextFlow està dissenyat per treballar amb text enriquit.
TilePane	Distribueix els seus fills en una quadrícula de "rajoles" de mida uniforme.
GridPane	Posa els nodes en cel·les en una graella bidimensional.
BorderPane	Posa els nodes en les regions centre, dreta, esquerra, sobre i sota definides pel panell.
HBox	Posa els nodes en una sola filera.
VBox	Posa els nodes en una sola columna.

48.12.1. *FlowPane*

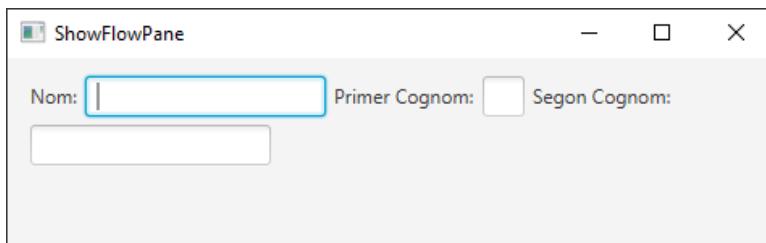
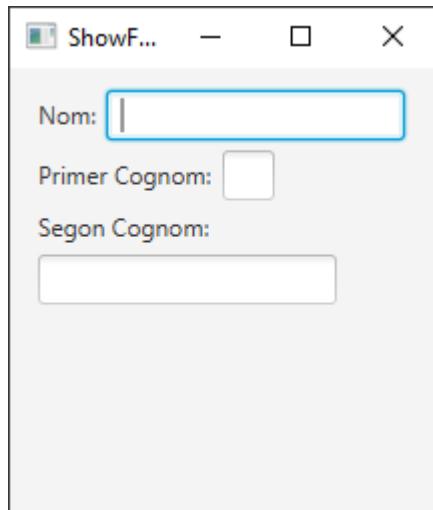
Posa els nodes un darrera l'altre canviant si cal de fila o columna.

Mètodes principals

C	FlowPane
<ul style="list-style-type: none"> □ alignment: ObjectProperty<Pos> {=Pos.LEFT} □ orientation: ObjectProperty<Orientation>{=Orientation.HORIZONTAL} □ hgap: DoubleProperty □ vgap: DoubleProperty 	
<ul style="list-style-type: none"> ● Flow Pane() ● Flow Pane(hgap: double, vgap: double) ● Flow Pane(orientation: ObjectProperty<Orientation>) ● Flow Pane(orientation: ObjectProperty<Orientation>, hgap: double, vgap: double) 	

Figura 48.9. javafx.scene.layout.FlowPane

Exemple



```
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.FlowPane;
import javafx.stage.Stage;

public class FlowPaneTest extends Application {

    @Override
    public void start(Stage primaryStage) {

        FlowPane pane = new FlowPane(); ①
        pane.setPadding(new Insets(11, 12, 13, 14)); ②
        pane.setHgap(5); ③
    }
}
```

```

        pane.setVgap(5); ④

        pane.getChildren().addAll(new Label("Nom:"), new
TextField(), new Label("Primer Cognom:"));
        TextField tfPrimerCognom = new TextField();
        tfPrimerCognom.setPrefColumnCount(1); ⑤
        pane.getChildren().addAll(tfPrimerCognom, new Label("Segon
Cognom:"), new TextField());

        Scene scene = new Scene(pane, 200, 250);
        primaryStage.setTitle("Test FlowPane");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}

```



Els controls apareixen en diferents posicions en funció de la mida del **Panel**.

- ①** Creem el FlowPane
- ②** Establim els marges interiors del Panell: top(11), right(12), bottom(13), left(14)
- ④** Establim la distància entre nodes.
- ⑤** Establim la mida de la caixa de text amb el mètode **setPrefColumnCount**.

48.12.2. TextFlow

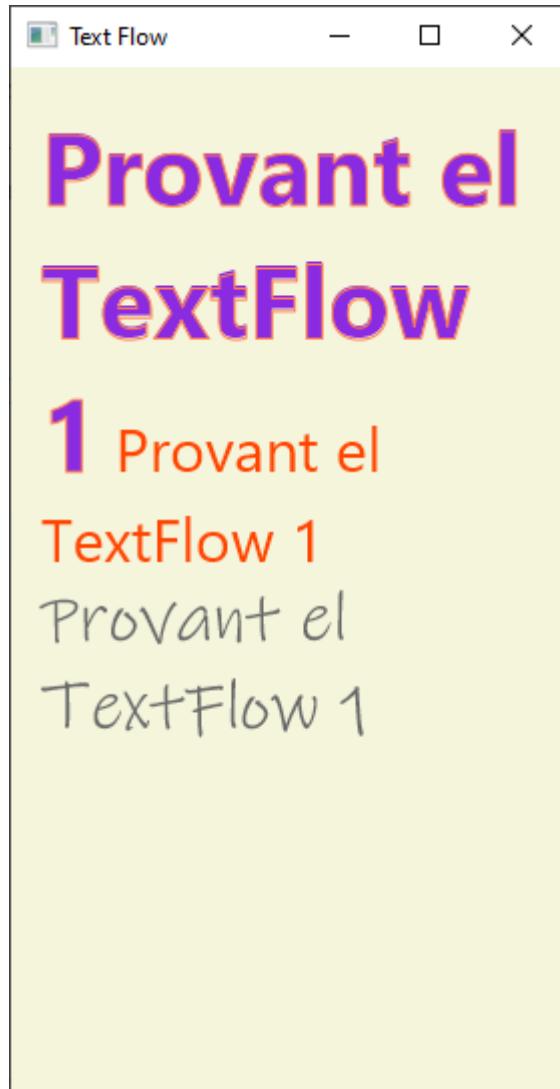
TextFlow està dissenyat per treballar amb text enriquit. Es pot utilitzar per gestionar diversos nodes de text en un sol flux de text. El **TextFlow** utilitza el text i el tipus de lletra de cada node de text que conté, a més de l'amplada i l'alignació del text, per determinar la ubicació de cada fill.

Qualsevol altre node, en lloc de text, es tractarà com a objecte incrustat a la disposició de text. S'inserirà al contingut mitjançant l'amplada, l'alçada i el desplaçament de línia de base preferits.

Els paràgrafs estan separats per `\n` present a qualsevol element secundari de text.

Exemple





```
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;
import javafx.scene.text.TextFlow;
import javafx.stage.Stage;
```

```
public class PaneDemo extends Application {  
  
    @Override  
    public void start(Stage primaryStage) {  
  
        Text text1 = new Text("Provant el TextFlow 1");  
        Font font1 = Font.font("Brush Script MT", FontWeight.BOLD, 50);  
        text1.setFont(font1);  
        text1.setFill(Color.BLUEVIOLET);  
        text1.setStrokeWidth(1);  
        text1.setStroke(Color.CORAL);  
  
        Text text2 = new Text(" Provant el TextFlow 1 ");  
        text2.setFont(new Font("Algerian", 30));  
        text2.setFill(Color.ORANGERED);  
  
        Text text3 = new Text("Provant el TextFlow 1");  
        Font font2 = Font.font("Ink Free", FontWeight.BOLD, 35);  
        text3.setFont(font2);  
        text3.setFill(Color.DIMGRAY);  
  
        TextFlow root = new TextFlow();  
  
        root.setPrefWidth(400);  
        root.getChildren().addAll(text1, text2, text3);  
  
        root.setPadding(new Insets(15, 15, 15, 15));  
  
        Scene scene = new Scene(root, 595, 250, Color.BEIGE);  
        primaryStage.setTitle("Text Flow");  
        primaryStage.setScene(scene);  
        primaryStage.show();  
    }  
  
    public static void main(String[] args) {  
        launch(args);  
    }  
}
```

48.12.3. TilePane

TilePane distribueix els seus fills en una quadricula de "rajoles" de mida uniforme.

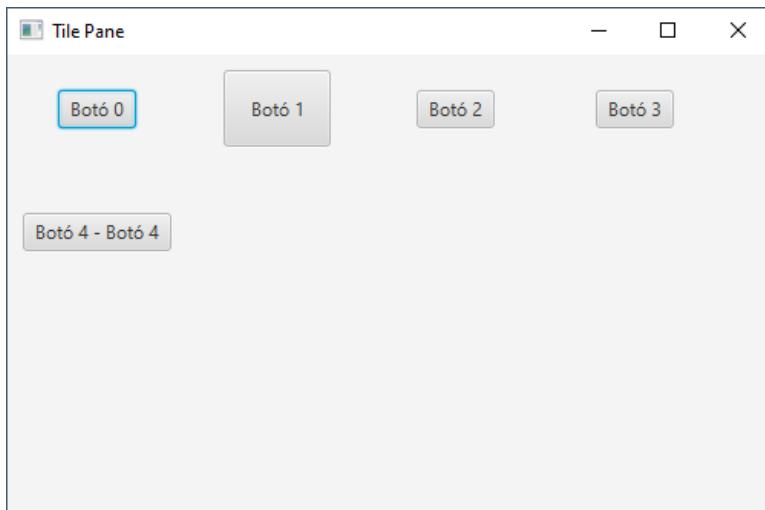
Un mosaic horitzontal (per defecte) mosaicarà els nodes de les files, ajustant-se a l'amplada del mosaic. Un mosaic vertical mosaicarà els nodes de les columnes i s'ajustarà a l'alçada del mosaic.

La mida de cada "rajola" és la mida necessària per incloure l'amplada i l'alçada preferides del fill més gran de les rajoles.

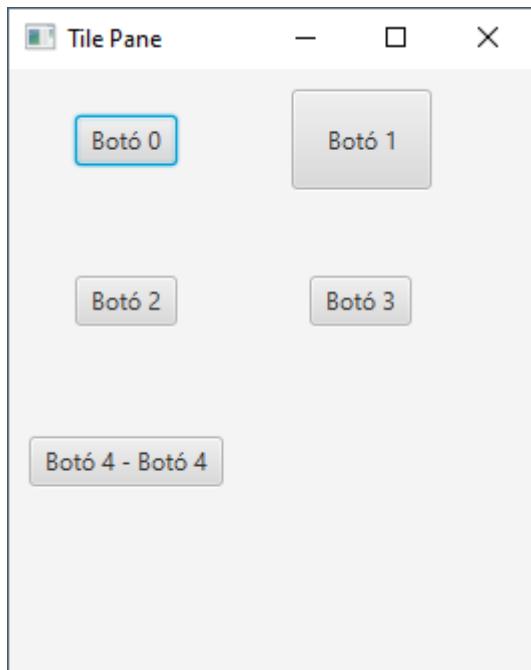
Les aplicacions haurien d'inicialitzar **prefColumns** (per a horitzontal) o **prefRows** (per a vertical) per establir la mida preferida del mosaic (el valor per defecte arbitrari és 5).

La propietat d'alignació controla com les files i les columnes s'alineen dins dels límits del mosaic i es defineixen per defecte a **Pos.TOP_LEFT**. També és possible controlar l'alignació dels nodes dins de les rajoles individuals establint **tileAlignment**, que per defecte és **Pos.CENTER**.

Exemple



TilePane



```
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.TilePane;
import javafx.stage.Stage;

public class PaneDemo extends Application {

    @Override
    public void start(Stage primaryStage) {

        TilePane root = new TilePane();

        root.setPadding(new Insets(10, 10, 10, 10));
        root.setHgap(20);
        root.setVgap(30);

        Button button0 = new Button("Botó 0");

        Button button1 = new Button("Botó 1");
        button1.setPrefSize(70, 50);
```

```

        Button button2 = new Button("Botó 2");

        Button button3 = new Button("Botó 3");

        Button button4 = new Button("Botó 4 - Botó 4");

        root.getChildren().addAll(button0, button1, button2, button3,
button4);

        Scene scene = new Scene(root, 500, 300);
        primaryStage.setTitle("Tile Pane");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}

```

48.12.4. GridPane

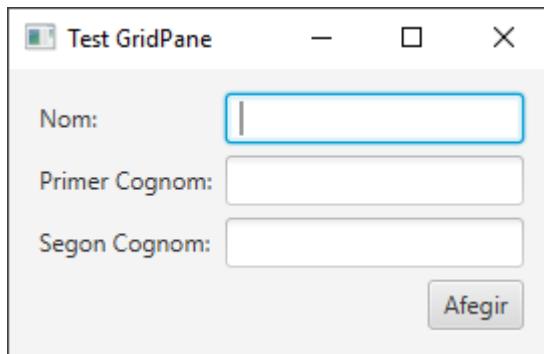
Un **GridPane** organitza els nodes en una graella. Els nodes es situen en coordenades específiques dins la graella.

Mètodes principals

 GridPane
<ul style="list-style-type: none"> □ alignment: ObjectProperty<Pos> {=Pos.LEFT} □ gridLinesVisible: BooleanProperty {=false} □ hgap: DoubleProperty {=0} □ vgap: DoubleProperty {=0} ● add(child: Node, columnIndex: int, rowIndex: int) ● addColumn(columnIndex: int, children: Node...) ● addRow (rowIndex: int, children: Node...) ● getColumnIndex(child: Node): int ● setColumnIndex(child: Node, columnIndex: int) ● getRowIndex(child: Node): int ● setRowIndex(child: Node, rowIndex: int) ● setHalignment(child: Node, value: HPos) ● setValignment(child: Node, value: VPos)

Figura 48.10. javafx.scene.layout.GridPane

Exemple



```

import javafx.application.Application;
import javafx.geometry.HPos;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;

public class GridPaneTest extends Application {

    @Override
    public void start(Stage primaryStage) {

        GridPane pane = new GridPane();
        pane.setAlignment(Pos.CENTER);
        pane.setPadding(new Insets(11.5, 12.5, 13.5, 14.5));
        pane.setHgap(5.5);
        pane.setVgap(5.5);

        // Place nodes in the pane
        pane.add(new Label("Nom:"), 0, 0); ①
        pane.add(new TextField(), 1, 0);
        pane.add(new Label("Primer Cognom:"), 0, 1);
        pane.add(new TextField(), 1, 1);
        pane.add(new Label("Segon Cognom:"), 0, 2);
        pane.add(new TextField(), 1, 2);
    }
}

```

```

        Button btAdd = new Button("Afegir");
        pane.add(btAdd, 1, 3);
        GridPane.setAlignment(btAdd, HPos.RIGHT);

        Scene scene = new Scene(pane);
        primaryStage.setTitle("Test GridPane");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}

```

- ① Especifiquem les coordenades de la cel·la a on posar el node.

48.12.5. BorderPane

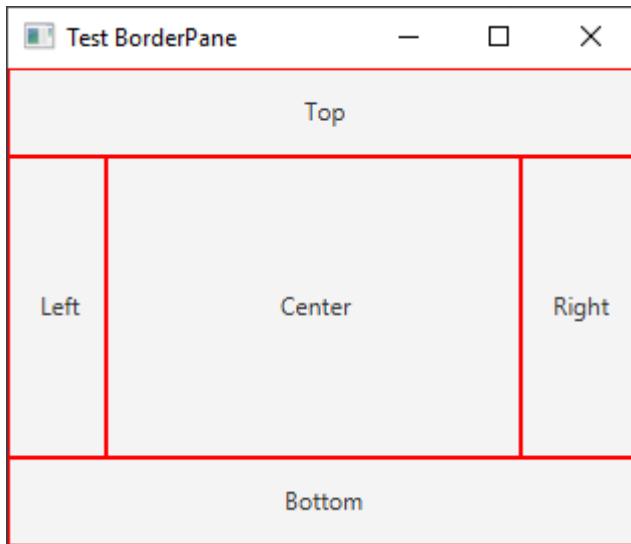
Posa els nodes en les regions centre, dreta, esquerra, sobre i sota definides pel panell.

Mètodes principals

C	BorderPane
<input type="checkbox"/> top: ObjectProperty<Node> <input type="checkbox"/> right: ObjectProperty<Node> <input type="checkbox"/> bottom: ObjectProperty<Node> <input type="checkbox"/> left: ObjectProperty<Node> <input type="checkbox"/> center: ObjectProperty<Node>	
<input checked="" type="checkbox"/> setAlignment(child: Node, pos: Pos)	

Figura 48.11. javafx.scene.layout.BorderPane

Exemple



```
public class BorderPaneTest extends Application {

    @Override
    public void start(Stage primaryStage) {

        BorderPane pane = new BorderPane();

        pane.setTop(new CustomPane("Top"));
        pane.setRight(new CustomPane("Right"));
        pane.setBottom(new CustomPane("Bottom"));
        pane.setLeft(new CustomPane("Left"));
        pane.setCenter(new CustomPane("Center"));

        Scene scene = new Scene(pane);
        primaryStage.setTitle("Test BorderPane");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

```
public class CustomPane extends StackPane {

    public CustomPane(String title) {
        getChildren().add(new Label(title));
        setStyle("-fx-border-color: red");
        setPadding(new Insets(11.5, 12.5, 13.5, 14.5));
    }
}
```

48.12.6. HBox i Vox

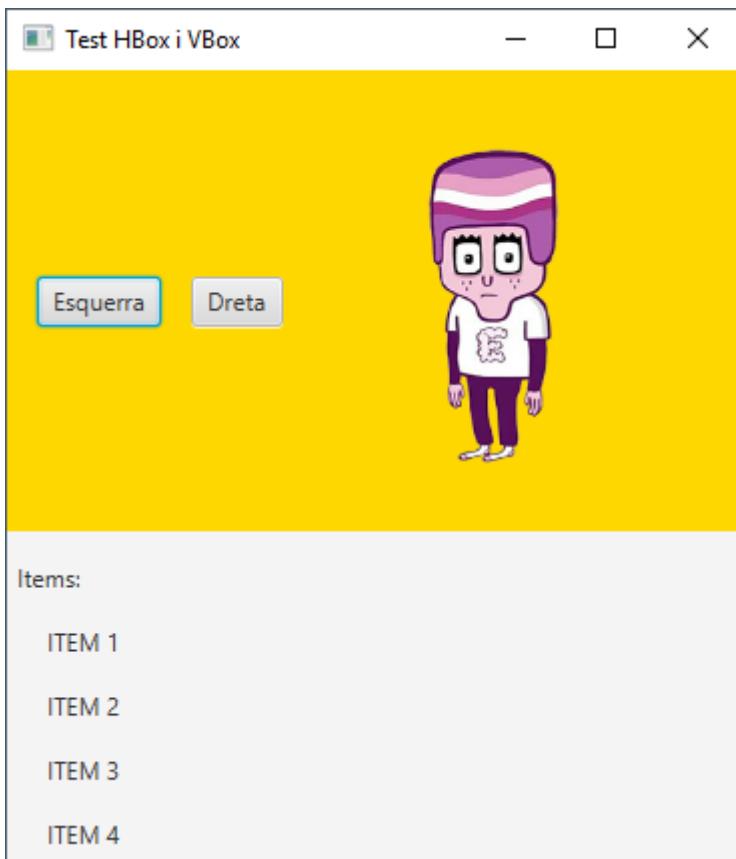
HBox posa els nodes en una sola filera horitzontal i VBox posa els nodes en una sola columna vertical.

Mètodes principals

	C HBox
	<ul style="list-style-type: none"> □ alignment: ObjectProperty<Pos> {=Pos.TOP_LEFT} □ fillHeight: BooleanProperty {=true} □ spacing: DoubleProperty {=0}
	<ul style="list-style-type: none"> ● HBox(spacing: double) ● setMargin(node: Node, value: Insets): void

Figura 48.12. javafx.scene.layout.HBox

Exemple



```
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class BorderPaneTest extends Application {
```

```
@Override
public void start(Stage primaryStage) {

    BorderPane pane = new BorderPane();

    pane.setTop(getHBox());
    pane.setLeft(getVBox());

    Scene scene = new Scene(pane);
    primaryStage.setTitle("Test HBox i VBox");
    primaryStage.setScene(scene);
    primaryStage.show();
}

private HBox getHBox() {
    HBox hBox = new HBox(15);
    hBox.setPadding(new Insets(15, 15, 15, 15));
    hBox.setStyle("-fx-background-color: gold");
    hBox.setAlignment(Pos.CENTER);
    hBox.getChildren().add(new Button("Esquerra"));
    hBox.getChildren().add(new Button("Dreta"));
    ImageView imageView = new ImageView(new Image("enjuto.png"));
    imageView.setFitHeight(200);
    imageView.setFitWidth(200);
    hBox.getChildren().add(imageView);
    return hBox;
}

private VBox getVBox() {
    VBox vBox = new VBox(15);
    vBox.setPadding(new Insets(15, 5, 5, 5));
    vBox.getChildren().add(new Label("Items:"));

    Label[] courses = {new Label("ITEM 1"), new Label("ITEM 2"),
        new Label("ITEM 3"), new Label("ITEM 4")};

    for (Label course : courses) {
        vBox.setMargin(course, new Insets(0, 0, 0, 15));
        vBox.getChildren().add(course);
    }

    return vBox;
}

public static void main(String[] args) {
```

```
    launch(args);  
}  
}
```

48.12.7. AnchorPane

Un **AnchorPane** permet fixar la posició dels seus nodes fills a una distància determinada de les seves vores.

Mètode principals

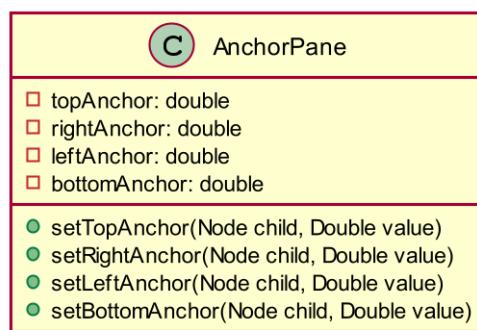
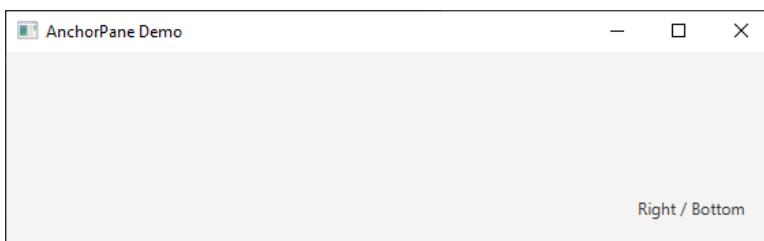


Figura 48.13. javafx.scene.layout.HBox

Exemple



```
package anchorpanedemo;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.AnchorPane;
import javafx.stage.Stage;

public class AnchorPaneDemo extends Application {

    @Override
    public void start(Stage primaryStage) {
        AnchorPane root = new AnchorPane();

        Label lbl1 = new Label("Right / Bottom");
        root.setRightAnchor(lbl1, 20d);
        root.setBottomAnchor(lbl1, 20d);

        root.getChildren().add(lbl1);

        Scene scene = new Scene(root, 300, 250);

        primaryStage.setTitle("AnchorPane Demo");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

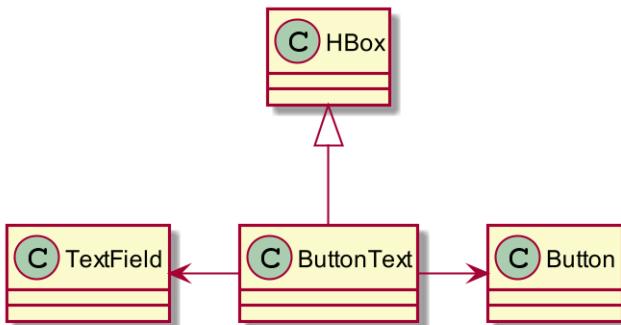
    public static void main(String[] args) {
        launch(args);
    }
}
```

48.13. Creació de nous controls

JavaFx ens permet la creació de nous controls essencialment seguint una de dues estratègies, **herència** o **composició**.

Vegem-ho directament amb dos exemples:

48.13.1. Crear un nou control per composició



```

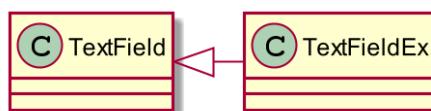
import javafx.scene.control.Button;
import javafx.scene.control.TextField;
import javafx.scene.layout.HBox;

public class ButtonText extends HBox {

    private Button btn;
    private TextField txt;

    public ButtonText(String bText, String tText) {
        this.btn = new Button(bText);
        this.txt = new TextField(tText);
        this.getChildren().addAll(btn, txt);
    }
}
  
```

48.13.2. Crear un nou control per herència



```

import javafx.scene.control.TextField;

public class TextFieldEx extends TextField {

    String numberRegEx = "[0-5]+";
  
```

```

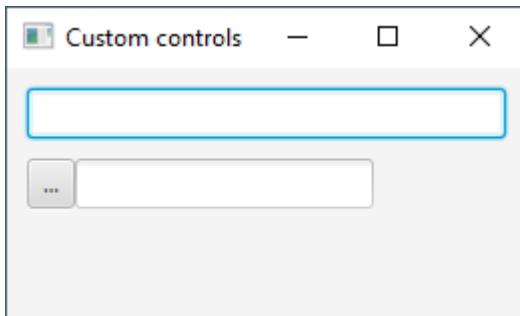
@Override
public void replaceText(int start, int end, String text) {
    if ((validate(text))) {
        super.replaceText(start, end, text);
    }
}

private boolean validate(String text) {

    return ("".equals(text) || text.matches(numberRegEx));
}
}

```

I podem provar els controls anteriors:



Main.

```

import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class App extends Application {

    @Override
    public void start(Stage stage) {
        VBox root = new VBox();
        root.setPadding(new Insets(10, 10, 10, 10));
        root.setSpacing(10);

        root.getChildren().add(new TextFieldEx());
    }
}

```

```
root.getChildren().add(new ButtonText("...", ""));  
Scene scene = new Scene(root, 200, 200);  
  
stage.setTitle("Custom controls");  
stage.setScene(scene);  
stage.show();  
}  
  
public static void main(String[] args) {  
    launch();  
}  
}
```

Inner classes i Nested classes

49.1. Què és una "nested class"

Una classe es pot declarar dins el cos d'una altra classe.

Per exemple:

Inner class.

```
package paquet1;

public class Outer {
    public class Inner {
        // Aquí van els membres de la classe interna
    }

    // Aquí van els membres de la classe Externa
}
```

Algunes peculiaritats de l'exemple anterior:

- La classe **Outer** és membre del paquet **paquet1**.
- La classe **Inner** és membre de la classe **Outer**.
- Una instància d'una "inner class" pot existir **només** dins d'una instància de la "outer class".



Les **nested class** es divideixen en dues categories: **static** i **no static**.

Les **nested class** declarades com a **static** s'anomenen **static nested class**.

Les **nested class** declarades com a **no static** s'anomenen **inner class**.

Una **static nested class** seria:

Static nested class.

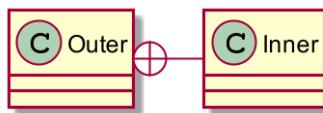
```
package paquet1;

public class Outer {
    public static class Nested {
        // Aquí van els membres de la classe interna
    }

    // Aquí van els membres de la classe Externa
}
```

49.2. Representació d'una classe "nested" en UML

Una *inner class* es pot representar en UML amb la notació següent:



49.3. Per què necessitem les nested class?

Bones raons per a la utilització de classes nested poden ser:

- Proporcionen una manera d'agrupar lògicament classes que només s'utilitzen en un sol lloc. Si una classe és útil només a una altre classe, té lògica posar la primera dins de la segona, això facilita la gestió del paquet en que es troben les classes.
- Incrementa l'encapsulació. Si tenim dues classes **A** i **B** i **B** necessita accés als membres de **A** aquests no pot ser declarats com a **private**, que és possiblement el que faríem. Encapsulant **B** dins de **A**, els membres de **A**

poden ser declarats com a privats i a més la classe **B** està amagada de l'exterior.

- Pot ajudar a obtenir codi més lleigible i més mantenible, és una bona regla de disseny intentar mantenir el codi prop del codi que l'utilitza, les "nested classes" ho fan de manera natural.

49.4. Tipus de "inner classes"

Es pot definir una "inner class" a qualsevol lloc on està permès posar una instrucció Java.

Les "inner class" poden ser de quatre tipus:

- "inner class" com a membre d'una altra classe
 - En aquest cas la classe pot ser declarada com **static nested class**.
 - o com a **no static inner class**.
- "inner class" local
- "inner class" anònima

49.4.1. "inner class" com a membre d'una altra classe

Una "inner class" declarada com a membre d'una altra classe es declara com qualsevol altre camp de la classe.

L'especificador d'accés pot ser **private**, **public**, **protected** o a **nivell-de-paquet**.

Característiques:

- La **inner class** té accés a tots els mètodes i camps de la **outer class**.
- Les instàncies de la **inner class** existeixen dins de les instàncies de la **outer class**.
- Per crear una instància d'una **inner class** cal fer-ho a través d'una instància de la **outer class**.
- Les **classes locals**

Per exemple:

```

public class Outer {

    private int outerValue;

    public void setOuterValue(int value) {
        outerValue = value;
    }

    public class Inner {

        private int innerValue;

        public void setInnerValue(int value) {
            innerValue = value;
        }

        public void setAllValues(int value) {
            innerValue = value;
            outerValue = value; // Podem accedir a les variables membre
de la classe externa
        }
    }
}

```

```

public class Main {

    public static void main(String[] args) {
        Outer out = new Outer();
        Outer.Inner in = out.new Inner();
    }
}

```

49.4.2. Classe "static nested class"

Si es declara la classe interna com a *static* obtenim una "static nested class".

En aquesta situació la classe interna no té accés als membres de la classe externa, caldria passar-li una referència.

```

public class Outer {

    private int outerValue;
}

```

```
public void setOuterValue(int value) {
    outerValue = value;
}

public static class Inner {

    private int innerValue;

    public void setInnerValue(int value) {
        innerValue = value;
    }
}
```

```
public class Main {

    public static void main(String[] args) {
        Outer.Inner in = new Outer.Inner();
    }
}
```

49.4.3. "inner class" local

Una "inner class" local es declara dins d'un bloc i només existeix dins del bloc on s'ha definit.

- No es poden utilitzar els especificadors d'accés **public**, **private** o **protected**.

```
class LocalInner{
    private String message = "Hola";
    void display(){
        class Local{
            void msg(){
                System.out.println(message);
            }
        }
        Local loc = new Local();
        loc.msg();
    }
    public static void main(String args[]){
        LocalInner obj = new LocalInner();
        obj.display();
    }
}
```

```
    }  
}
```

49.4.4. "inner class" anònima

Una classe anònima és una classe definida dins d'una altra classe, una *inner class*, amb la particularitat que no té nom.

Com que no té nom **no pot tenir constructor**.

Aleshores com es poden crear objectes d'aquesta classe?

La única manera és crear un objecte just en el moment en que es defineix la classe, només es podrà crear un objecte a partir d'una classe anònima.

La sintaxi per crear un objecte a partir d'una classe anònima és:

```
new <nom-interficie | nom-classe> (<llista-arguments> {  
    // Cos de la classe anònima  
}
```

El *nom-interficie* o *nom-classe* és una classe existent, **no és el nom de la classe anònima**, la classe anònima implementa la interfície o hereta de la classe indicada.

Per exemple:

```
interface Interficie {  
    void metode()  
}  
  
class Classe {  
  
    private void crearClasseAnonima() {  
        Interficie classe = new Interficie() {  
            void metode() {  
                System.out.println("Hola! !");  
            }  
        };  
  
        classe.metode();  
    }  
}
```

{}

49.5. Exemple 1 - Inner class

```
package exempleinnerclass;

public class DataStructure {

    private final static int MIDA = 15;
    private int[] enters = new int[MIDA];

    public DataStructure() {
        for (int i = 0; i < MIDA; i++) {
            enters[i] = i;
        }
    }

    public DataStructureIterator getIterator() {
        return new SenarIterator();
    }

    interface DataStructureIterator extends java.util.Iterator<Integer>
    {
    }

    private class SenarIterator implements DataStructureIterator {

        private int seguentIndex = 0;

        public boolean hasNext() {

            // Comprovem si és l'últim element de la matriu
            return (seguentIndex <= MIDA - 1);
        }

        public Integer next() {

            Integer retValue = Integer.valueOf(enters[seguentIndex]);

            seguentIndex += 2;
            return retValue;
        }
    }
}
```

```

public static void main(String s[]) {

    DataStructure ds = new DataStructure();
    DataStructure.DataStructureIterator iterator = ds.getIterator();

    while (iterator.hasNext()) {
        System.out.print(iterator.next() + " ");
    }

    System.out.println();
}
}

```

49.6. Exemple 2 - Local class

```

package exempleclasselocal;

public class ExempleClasseLocal {

    public static String validarNumeroTelefon(String numTelefon) {
        String regexNoEsUnDigit = "[^0-9]";
        int longitudNumeroTelefon = 9;

        class NumeroTelefon {

            String numeroTelefonAmbFormat = null;

            NumeroTelefon(String numeroTelefon) {
                String numeroTelefonAmbFormat =
numeroTelefon.replaceAll(regexNoEsUnDigit, "");
                if (numeroTelefonAmbFormat.length() ==
longitudNumeroTelefon) {
                    this.numeroTelefonAmbFormat =
numeroTelefonAmbFormat;
                } else {
                    this.numeroTelefonAmbFormat = null;
                }
            }

            public String getNumeroTelefon() {
                return numeroTelefonAmbFormat;
            }
        }
    }
}

```

```

    NumeroTelefon num = new NumeroTelefon(numTelefon);

    return num.getNumeroTelefon();
}

public static void main(String... args) {
    // String numOrig="123-456-7890";
    String numOrig = "(93) 719 3232";
    System.out.print(numOrig + " -> ");
    String num = validarNumeroTelefon(numOrig);
    if (num != null) {
        System.out.println(num);
    } else {
        System.out.println("Invàlid!!!");
    }
}
}

```

49.7. Exemple 3 - Classe anònima

```

package exempleclasseanonima;

public class HolaMonClassesAnonimes {

    interface HolaMon {

        public void saludar();

        public void saludar(String algu);
    }

    public void dirHola() {

        class SaludarEnAngles implements HolaMon {

            String nom = "world";

            public void saludar() {
                saludar("world");
            }

            public void saludar(String algu) {
                nom = algu;
                System.out.println("Hello " + nom);
            }
        }
    }
}

```

```
        }

    }

HolaMon saludarEnAngles = new SaludarEnAngles();

HolaMon saludarEnFrances = new HolaMon() {
    String nom = "tout le monde";

    public void saludar() {
        saludar("tout le monde");
    }

    public void saludar(String algu) {
        nom = algu;
        System.out.println("Salut " + nom);
    }
};

HolaMon saludarEnCatala = new HolaMon() {
    String name = "món";

    public void saludar() {
        saludar("món");
    }

    public void saludar(String someone) {
        name = someone;
        System.out.println("Hola, " + name);
    }
};

saludarEnAngles.saludar();
saludarEnFrances.saludar("Fred");
saludarEnCatala.saludar();
}

public static void main(String... args) {
    HolaMonClassesAnonimes app = new HolaMonClassesAnonimes();
    app.dirHola();
}
}
```

50

Interfícies funcionals

50.1. Com encapsular una funció en una variable

L'objectiu que perseguiu és poder encapsular una funció dins d'una variable.

Això ens permetria poder tenir, per exemple, matrius de funcions, mètodes que admetin funcions com a paràmetres o mètodes que retornin una funció.

Imaginem que volem un mètode de nom **metode** que admeti una funció com a paràmetre, una cosa així:

```
void metode(Funcio f) {  
    ...  
}
```

50.2. Proposta solució - versió 1

Com que tradicionalment en Java les variables només poden contenir objectes o valors de tipus primitius, si el que es vol és posar-hi una funció es pot fer la construcció següent:

```
class EncapsulaFuncio {  
    public void funcio() {  
        ...  
    }  
}
```

I si volem un mètode que admeti una funció com a paràmetre faríem:

```
class UnaClasse {
    void metode(EncapsulaFuncio encfunc) {
        // Implementació de la funció
    }
}
```

Encara podríem fer-ho millor declarant una interfície, l'exemple anterior quedaría:

50.3. Proposta solució - versió 2

```
interface Funcio {
    void funcio();
}
```

```
class EncapsulaFuncio implements Funcio {
    @Override
    public void funcio() {
        // Implementació de la funció
    }
}
```

I si volem un mètode que admeti una funció com a paràmetre faríem:

```
void metode(Funcio encfunc) {
    ...
}
```

Amb la implementació anterior, cada cop que es vulgui canviar la funció **funcio()** caldrà crear una nova classe que implementi la interfície **Funcio** i programar la nova implementació de la funció, per exemple, si volem disposar de dues funcions diferents caldria fer:

```
class EncapsulaFuncio1 implements Funcio {
    @Override
    public void funcio() {
        // Implementació de la funció 1
    }
}
```

```
class EncapsulaFuncio2 implements Funcio {
    @Override
    public void funcio() {
        // Implementació de la funció 2
    }
}
```

No obstant, utilitzant classes anònimes ens podríem evitar declarar una nova classe cada cop que volguéssim tenir una nova funció:

50.4. Proposta solució - versió 3

```
Funcio f = new Funcio() {
    @Override
    public void funcio() {
        // Implementació de la funció
    }
};
```

50.5. Interfícies funcionals

A partir de Java 8, apareix el concepte d'**interfície funcional**, una interfície funcional és una interfície amb un **únic mètode abstracte**, anomenat **mètode funcional** decorada amb l'anotació **@FunctionalInterface**.

Per exemple:

Interfície funcional.

```
@FunctionalInterface
interface Funcio
{
    String funcio(String x);
}
```

Si la interfície **Funcio** contingüés més d'un mètode abstracte, l'anotació **@FunctionalInterface** generaria un error de compilació.



Per ser una **interfície funcional** es demana tenir un únic mètode **abstracte**, les interfícies funcionals poden tenir

mètodes **default** i mètodes **static** a part del mètode funcional.

50.6. Mètodes **default** a les interfícies funcionals

Una interfície funcional pot proporcionar mètodes **default**, un mètode **default** definit en una interfície és un mètode que proporciona una implementació que opcionalment es pot sobreescrivir en una interfície derivada o a la implementació de la interfície.

Vegem un exemple:

1. Definim una interfície funcional amb un mètode **default**:

```
@FunctionalInterface  
interface Funcio {  
  
    String funcio(String x);  
  
    default void showString(String s) {  
        System.out.println(s);  
    }  
}
```

2. Creem una classe i implementem la interfície anterior, fixeu-vos que no estem implementant el mètode declarat com a **default**.

```
class TestInterface implements Funcio {  
  
    @Override  
    public String funcio(String x) {  
        return x.toUpperCase();  
    }  
}
```

3. Finalment creem una instància de la classe que implementa la interfície i comprovem que es poden cridar els dos mètodes que defineix.

```
public class Main {  
  
    public static void main(String[] args) {
```

```
    TestInterface ti = new TestInterface();
    ti.showString(ti.funcio("test"));
}
}
```

50.7. Mètodes static a les interfícies funcionals

Una interfície funcional també pot tenir mètodes **static**. Els mètodes **static** són útils per definir mètodes "helper" amb la implementació a la pròpia interfície però que no estan pensats per ser sobreescrits en les classes que implementen la interfície.

Per exemple, el mètode **default** de l'exemple anterior el podríem convertir a un mètode **static**. Vegem-ho:

1. Definim una interfície funcional amb un mètode **static**:

```
@FunctionalInterface
interface Funcio {

    String funcio(String x);

    static void showString(String s) { ❶
        System.out.println(s);
    }
}
```

2. Creem una classe i implementem la interfície anterior, fixeu-vos que no estem implementant el mètode declarat com a **static**.

```
class TestInterface implements Funcio {

    @Override
    public String funcio(String x) {
        return x.toUpperCase();
    }
}
```

3. Finalment creem una instància de la classe que implementa la interfície, el mètode proporcionat per la interfície el cridem a partir d'un objecte i el mètode **static** el cridem des de la pròpia interfície directament.

```
public class Lambdes {  
  
    public static void main(String[] args) {  
        TestInterface ti = new TestInterface();  
        Funcio.showString(ti.funcio("test")); ①  
    }  
}
```

51

Expressions Lambda

Si la nostra aplicació necessita moltes implementacions d'interfícies funcionals caldrà **implementar una gran quantitat de classes anònimes** o no dificultant la lectura i la mantenibilitat del codi.

A partir de Java 8 el llenguatge proporciona unes construccions, anomenades **expressions lambda** que permeten representar **interfícies funcionals** d'una manera molt compacte.

La sintaxi general d'una **expressió lambda** és la següent:

```
llista_arguments -> cos_de_la_expressió
```

on

cos_de_la_expressió

Proporciona la implementació del mètode funcional.

llista_arguments

Proporciona els arguments passats al mètode funcional.

Per exemple, la interfície funcional següent:

```
@FunctionalInterface  
interface IFuncio {  
    String funcio(String s);  
}
```

Es pot representar per la següent expressió lambda:

```
s -> s
```

La següent sentència assigna l'expressió lambda a un objecte d'una classe que implementa la interfície funcional **IFuncio**:

```
IFuncio lambda = x -> x;
```

Quan es crida el mètode funcional de l'objecte, s'executa l'expressió lambda passant "Hola" al paràmetre **x**, l'expressió s'avalua i es proporciona "Hola" com a valor de retorn.

```
System.out.println(lambda.funcio("Hola"));
```

Si el mètode funcional representat per l'expressió lambda retorna **void**, el cos de l'expressió lambda ha de ser una expressió que retorni **void**, per exemple:

```
@FunctionalInterface
interface IFuncio {
    void funcio(int i);
}
```

```
IFuncio lambda = x -> System.out.println(x);
lambda.funcio(10);
```

51.1. Estructura de la llista d'arguments

Si la llista d'arguments té un sol argument i el **tipus es pot inferir**, l'argument es pot especificar sense parèntesis:

```
x -> x
```

Es pot especificar un tipus per l'argument de la següent manera:

```
(String s) -> s
```

Si hi ha més d'un argument, cadascun d'ells es separa amb una coma, per exemple:

Donada la següent interfície funcional:

```
@FunctionalInterface  
interface IFuncio {  
    String funcio(String s, int i);  
}
```

Tindríem:

```
IFuncio lambda = (s, i) -> s + Integer.toString(i+10);
```

o bé

```
Funcio lambda = (String s, int i) -> s + Integer.toString(i+10);
```

Si la funció no té arguments es pot especificar amb un parell de parèntesis buits, per exemple.

```
@FunctionalInterface  
interface IFuncio {  
    Integer funcio();  
}
```

```
IFuncio f = () -> 10
```

51.2. Estructura del cos de la expressió lambda

El cos d'una funció lambda es pot declarar com una expressió o com un bloc:

Com una expressió:

```
IFuncio f = s -> System.out.println("Hola");
```

com un bloc:

```
IFuncio f = s -> {
```

```

System.out.println("Hola");
System.out.println("Hola");
System.out.println("Hola");
};

```

51.3. Abast d'una expressió Lambda

L'abast d'una expressió Lambda coincideix amb l'abast de la expressió que la defineix, es a dir, **qualsevol variable accessible des del punt on es defineix l'expressió també és accessible des del mètode funcional que representa.** Per exemple:

```

interface IFuncio {

    void funcio(int x);
}

public class TestAbastLambda {

    private int camp;

    public TestAbastLambda(int valor) {
        this.camp = valor;
    }

    public static void main(String[] args) {

        TestAbastLambda o = new TestAbastLambda(10);
        o.crearLambda();
    }

    public void crearLambda() {
        int local = 100;
        // local++ ❶
        IFuncio lambda = x -> System.out.println(x + local + camp);
        lambda.funcio(1000);
    }
}

```

- ❶ Les variables locals utilitzades en expressions lambda, a la pràctica, han de ser **final**, la línia senyalada donaria un error de compilació.

51.4. Predicats

A Java 8, el paquet **java.util.function** conté una sèrie d'interfícies funcionals dissenyades per ajudar en la programació funcional.

Aquestes interfícies funcionals segueixen un dels quatre models següents:

Taula 51.1. Models bàsics de les interfícies funcionals de java.util.function

Interfície	Té arguments	Retorna un valor	Descripció
Predicate	sí	boolean	Comprova els arguments i retorna true o false .
Function	sí	sí	Mapeja un tipus a un altre.
Consumer	sí	no	Consumeix l'entrada.
Supplier	no	sí	Genera sortida.

La definició de les interfícies anteriors és la següent:

Predicate.

```
@FunctionalInterface
public interface Predicate<T> {
    default Predicate<T> and(Predicate<? super T> other);
    default Predicate<T> negate();
    default Predicate<T> or(Predicate<? super T> other);
    static <T> Predicate<T> isEqual(Object targetRef);
    boolean test(T t);
}
```

Function.

```
@FunctionalInterface
public interface Function<T,R> {
    default <V> Function<T,V> andThen(Function<? super R, ? extends V>
after);
    default <V> Function<V,R> compose(Function<? super V, ? extends T>
before);
    static <T> Function <T,T> identity();
    R apply(T t);
}
```

Supplier.

```
@FunctionalInterface  
public interface Supplier<T> {  
    T get();  
}
```

Consumer.

```
@FunctionalInterface  
public interface Consumer<T> {  
    void accept(T t);  
    default Consumer<T> andThen(Consumer<? super T> after);  
}
```

51.5. Exemples

51.5.1. Exemple 1

```
Predicate<Integer> p1 = x -> x > 7;  
System.out.println(p1.or(x -> x < 3).test(9));
```

```
true
```

51.5.2. Exemple 2

```
System.out.println(((Predicate<Integer>) (x -> x > 7)).or(x -> x  
< 3).test(9)); ①
```

```
true
```

51.5.3. Exemple 3

```
Predicate<Integer> p1 = x -> x > 7;  
System.out.println(p1.or(Predicate.isEqual(3)).test(3)); ①
```

```
true
```

51.5.4. Exemple 4

```
Function<String, Integer> f = x -> Integer.parseInt(x);
System.out.println(f.apply("250"));
```

250

51.5.5. Exemple 5

```
Function<String, Boolean> f = x -> Boolean.parseBoolean(x);
System.out.println(f.andThen(x -> x == true ? 1 : 0).apply("true"));
```

1

52

Exemple - Classes anònimes i Lambdes

Suposem que estem creant una aplicació de gestió d'un club de lectura i volem implementar una funcionalitat que permeti a un administrador fer una determinada acció, enviar un missatge per exemple, als membres donats d'alta a l'aplicació que satisfacin un determinat criteri.

El cas d'ús seria el següent:

Nom del cas d'ús	UC1 - Fer una acció sobre els membres seleccionats
Actors participants	Administrador
Activació	A criteri de l' Administrador
Precondicions	Administrador ha fet login al sistema
Postcondicions	L'acció només es realitza sobre els membres que compleixen un determinat criteri.

Escenari principal		
Seq.	Actor	Sistema
1	L' Administrador especifica el criteri dels membres sobre els que efectuar determinada acció.	

2	L' Administrador prem el botó Acceptar	
3		El Sistema troba tots els membres que compleixen el critèri especificat.
4	El Sistema efectua l'acció especificada sobre tots els membres que compleixen el critèri especificat.	

Extensions

1a	[Administrador sel·lecciona el botó Previsualització] → L' Administrador té la opció de previsualitzar els membres que compleixen el criteri especificat abans de prémer el botó Acceptar .	
----	--	--

Suposem que els membres del club de lectura estan representats per la següent classe:

```
package clubdelectura;

import java.time.LocalDate;
import java.time.Period;

public class Membre {

    public enum Genere {
        HOME, DONA
    }

    private String nom;
    private LocalDate dataNaixement;
    private Genere genere;

    public Membre(String nom, LocalDate dataNaixement, Genere genere) {
        this.nom=nom;
```

```

        this.dataNaixement=dataNaixement;
        this.genere=genere;
    }

    public int getEdat() {
        return Period.between(getDataNaixement(),
LocalDate.now()).getYears();
    }

    @Override
    public String toString() {
        return String.format("%s (%s) nascut al %s (%d anys)", getNom(),
getGenere().toString(), getDataNaixement().toString(),getEdat());
    }

    public String getNom() {
        return nom;
    }

    public LocalDate getDataNaixement() {
        return dataNaixement;
    }

    public Genere getGenere() {
        return genere;
    }
}

```

I podem fer servir la següent **ArrayList** per a proves.

```

public static void main(String[] args) {

    List<Membre> membres = new ArrayList<>();

    membres.add(new Membre("JOSE MANUEL MARTORELL MOLLA",
LocalDate.of(1970, Month.MARCH, 3), Membre.Genere.HOME));
    membres.add(new Membre("SUSANA RIESCO SERNA", LocalDate.of(1945,
Month.APRIL, 13), Membre.Genere.DONA));
    membres.add(new Membre("JOSE LORENTE CARRASCOSA", LocalDate.of(1998,
Month.JANUARY, 24), Membre.Genere.HOME));
    membres.add(new Membre("AURORA DOMENECH CASCALES",
LocalDate.of(2000, Month.JULY, 28), Membre.Genere.DONA));
    membres.add(new Membre("IGNACIO GELABERT ALMENDROS",
LocalDate.of(1979, Month.OCTOBER, 19), Membre.Genere.HOME));
}

```

Aproximació 1: Crear mètodes que busquin membres per una característica

```
membres.add(new Membre("AURORA DOMENECH CASCALES",
LocalDate.of(1989, Month.JANUARY, 1), Membre.Genere.DONA));
membres.add(new Membre("HUGO SOBRINO COMESAÑA", LocalDate.of(2008,
Month.MARCH, 2), Membre.Genere.HOME));
membres.add(new Membre("ROSA CANDEL LLACER", LocalDate.of(1993,
Month.DECEMBER, 3), Membre.Genere.DONA));
membres.add(new Membre("SAMUEL PRADOS ARNEDO", LocalDate.of(2012,
Month.MARCH, 3), Membre.Genere.HOME));
membres.add(new Membre("SUSANA SAN EMETERIO TORIBIO",
LocalDate.of(2002, Month.MARCH, 30), Membre.Genere.DONA));
}
```

Suposem que els membres del club s'emmagatzemen en una instància de **List<Membre>** i volem implementar diferents mètodes de cerca en funció de les diferents característiques dels membres.

52.1. Aproximació 1: Crear mètodes que busquin membres per una característica

Una manera de procedir podria ser crear un mètode per a cada característica de cerca, major d'una edat, gener, etc...

Per exemple:

```
public static void mostrarMembresMesVellsQue(List<Membre> membres, int
edat) {
    for (Membre p : membres) {
        if (p.getEdat() >= edat) {
            p.printMembre();
        }
    }
}
```

Aquesta aproximació és poc robusta, un canvi a l'estructura de **Membre** obligaria a reescriure una gran part dels mètodes de cerca existents. A més, es poc flexible, cal un mètode per cadascun del criteris de cerca possibles cosa que pot implicar modificacions freqüents al codi a mesura que van apareixen necessitats de filtres nous.

52.2. Aproximació 2: Crear mètodes de cerca més generals

El següent mètode és més genèric que l'anterior, mostra els membres dins d'un rang d'edats.

```
public static void mostrarMembresAmbEdatEntre(List<Membre> membres, int low, int high) {
    for (Membre m : membres) {
        if (low <= m.getEdat() && m.getEdat() < high) {
            System.out.println(m);
        }
    }
}
```

El problema segueix essent el mateix, el mètode no és prou flexible per fer cerques diferents, per genere o per mes de naixement per exemple.

Podríem anar afegint diferents mètodes per a diferents cerques, però això complicaria el manteniment del programa en cas, per exemple, d'un canvi a la classe **Membre**.

52.3. Aproximació 3: Especificar el codi del criteri de cerca en una classe local

Ens agradaria tenir un únic mètode de cerca al que se li poguessin passar uns criteris de manera que el mètode mostres únicament aquells membres que complissin els criteris passats.

Per tant, necessitem crear una classe que encapsuli els criteris de cerca i un mètode que el reculli i filtri en funció d'aquests.

El mètode passarà a ser una cosa similar al següent:

```
public static void mostrarMembres(List<Membre> membres,
ComprovarSiMembreOptaDescompteJove filtre) {
    for (Membre m : membres) {
        if (filtre.comprovar(m)) {
            System.out.println(m);
        }
    }
}
```

Aproximació 4: Especificar el codi
del criteri de cerca en una interfície

```
}
```

I una classe **ComprovarMembre** que encapsuli la condició de filtratge.

```
class ComprovarSiMembreOptaDescompteJove {  
  
    public boolean comprovar(Membre m) {  
        return m.getEdat() >= 18 && m.getEdat() <= 25;  
    }  
}
```

52.4. Aproximació 4: Especificar el codi del criteri de cerca en una interfície

Volem generalitzar el codi anterior, una manera de fer-ho és encapsular el mètode **comprovar** dins d'una interfície, d'aquesta manera no caldrà crear un mètode **mostrarMembres** per a cada condició del filtre.

```
interface Filtre {  
  
    boolean comprovar(Membre m);  
}
```

```
class ComprovarSiMembreOptaDescompteJove implements Filtre {  
  
    @Override  
    public boolean comprovar(Membre m) {  
        return m.getEdat() >= 18 && m.getEdat() <= 25;  
    }  
}
```

```
public static void mostrarMembres(List<Membre> membres, Filtre filtro) {  
    for (Membre m : membres) {  
        if (filtro.comprovar(m)) {  
            System.out.println(m);  
        }  
    }  
}
```

i cridariem el mètode de la següent manera:

Aproximació 5: Especificar el codi del criteri de cerca en una classe anònima

```
mostrarMembres(membres, new ComprovarSiMembreOptaDescompteJove());
```

52.5. Aproximació 5: Especificar el codi del criteri de cerca en una classe anònima

Arribats a aquests punt ens fixem que tenim totes les condicions per treballar amb una classe anònima enlloc de una classe concreta com ara **ComprovarSiMembreOptaDescompteJove**.

Podem fer el següent:

```
mostrarMembres(membres, new Filtre() {  
    @Override  
    public boolean comprovar(Membre m) {  
        return m.getEdat() >= 18 && m.getEdat() <= 25;  
    }  
});
```

52.6. Aproximació 6: Especificar el codi del criteri de cerca en una expressió Lambda

Equivalentment al punt anterior podem substituir la classe anònima per una expressió Lambda.

```
mostrarMembres(membres, (Membre m) -> m.getEdat() >= 18 && m.getEdat()  
<= 25);
```

52.7. Aproximació 7: Utilitzar interfícies funcionals estàndards i expressions Lambda

Reconsiderem la interfície **Filtre**.

```
interface Filtre {  
  
    boolean comprovar(Membre m);  
}
```

És una interfície funcional ja que només conté un únic mètode i a més té la mateixa signatura que la interfície **Predicate<Membre>** de **java.util.Function**.

Per tant podem refactoritzar el nostre codi i utilitzar **Predicate<Membre>** enlloc de **Filtre**.

```
public static void mostrarMembres(List<Membre> membres,
Predicate<Membre> filtre) {
    for (Membre m : membres) {
        if (filtre.test(m)) {
            System.out.println(m);
        }
    }
}
```

52.8. Aproximació 8: Utilitzar expressions Lambda a tota l'aplicació

Arribats aquest punt encara podem fer alguna millora. D'entrada podem passar el filtre que s'aplicarà a la llista de membres, però podríem generalitzar encara més el mètode si a més poguéssim passar quina acció s'ha de realitzar amb els membres que passen el filtre.

De moment el que fem amb els membres que passen el filtre és mostrar-ho per pantalla, podríem considerar que l'acció que volem fer amb els membres seleccionats és una acció que no retorna res, en termes dels predicats de la programació funcional en Java estaríem parlant d'un **Consumer<T>**.

El mètode anterior amb la nova notació quedaria:

```
public static void processarMembres(List<Membre> membres,
Predicate<Membre> filtre, Consumer<Membre> accio) {
    for (Membre m : membres) {
        if (filtre.test(m)) {
            accio.accept(m);
        }
    }
}
```

```
processarMembres(membres, m -> m.getEdat() >= 18 && m.getEdat() <= 25, m  
-> System.out.println(m));
```

52.9. Aproximació 9: Introduir genèrics

Finalment podríem generalitzar el mètode anterior perquè funcionés per a llistes de qualsevol tipus d'elements, és a dir, podríem introduir els genèrics. Quedaria una cosa similar a la següent:

```
public static <E> void processarElements(List<E> elements, Predicate<E>  
filtre, Consumer<E> accio) {  
    for (E m : elements) {  
        if (filtre.test(m)) {  
            accio.accept(m);  
        }  
    }  
}
```


53

Fluxos

Les col·leccions ens permeten iterar sobre una llista d'elements i ordenar, seleccionar o recuperar elements, és perfectament vàlid treballar d'aquesta manera, no obstant, quan es treballa amb un gran volum de dades, iterar sobre els elements de la col·lecció és un procés lent i poc eficient. Per aquesta raó a partir de Java 8 existeix una API específica, la **java.util.stream** que permet processar col·leccions utilitzant les propietats de multitasca de les màquines actuals.

El processament d'un flux de dades es realitza en tres etapes:

1. Crear un objecte **Stream** des de zero o bé extreure'l d'una **matriu**, una **col·lecció**, una **funció generadora**, un **canal d'E/S**, etc...
2. Especificar operacions sobre el flux de dades transformant el flux inicial en d'altres fluxos de dades.
3. Aplicar una operació **terminal** i produir un resultat. Aplicar una operació terminal força l'operació de les operacions precedents i evita que el flux pugui ser utilitzat més endavant.



A més de **Stream**, que és un flux de referències d'objectes, hi ha especialitzacions primitives per a **IntStream**, **LongStream** i **DoubleStream**, totes elles anomenades "fluxos" i que s'ajusten a les característiques i restriccions aquí descrites.

Per exemple:

```
int sum = widgets.stream()
```

```
.filter(w -> w.getColor() == RED)
.mapToInt(w -> w.getWeight())
.sum();
```

En aquest exemple, els **widgets** són una col·lecció <**Widget**>. Creem un flux d'objectes **Widget** mitjançant **Collection.stream()**, el filtram per produir un flux que conté només els widgets vermells i, a continuació, el transformem en un flux de valors *int* que representen el pes de cada widget vermell. A continuació, aquest flux es suma per produir un pes total.

Un aspecte important del processament dels fluxos és que **les operacions intermèdies no s'executen fins que s'invoca una operació terminal**, cada operació intermèdia crea un nou flux i s'emmagatzema junt amb la funció proporcionada, quan es crida a una operació terminal es crida a cadascuna de les funcions una a una fins a obtenir el resultat.

Cal tenir en compte que la API **Stream** és declarativa, no imperativa, això vol dir que ens preocupem del **què** s'ha de fer enllloc del **com** s'ha de fer que és l'enfoc habitual en la programació estructurada.

53.1. Iteracions vs Streams

Per veure com funciona tot plegat mireu el següent exemple on s'implementa el mateix programa amb col·leccions i a continuació amb fluxos.

53.1.1. Exemple - Cerca de productes versió Col·leccions

```
public class Producte {

    private String producteId;
    private String fabricant;
    private String nomProducte;
    private double preuUnitari;

    public Producte(String producteId, String fabricant, String
nomProducte, double preuUnitari) {
        this.producteId = producteId;
        this.fabricant = fabricant;
        this.nomProducte = nomProducte;
        this.preuUnitari = preuUnitari;
    }
}
```

```
public Producte() {  
}  
  
public String getProductId() {  
    return productId;  
}  
  
public void setProductId(String productId) {  
    this.productId = productId;  
}  
  
public String getFabricant() {  
    return fabricant;  
}  
  
public void setFabricant(String fabricant) {  
    this.fabricant = fabricant;  
}  
  
public String getNomProducte() {  
    return nomProducte;  
}  
  
public void setNomProducte(String nomProducte) {  
    this.nomProducte = nomProducte;  
}  
  
public double getPreuUnitari() {  
    return preuUnitari;  
}  
  
public void setPreuUnitari(double preuUnitari) {  
    this.preuUnitari = preuUnitari;  
}  
  
public String toString() {  
    return productId + " " + fabricant + " " + nomProducte + " " +  
    preuUnitari;  
}
```

```
import java.util.ArrayList;  
import java.util.List;
```

```
public class SenseStreams {

    public static void main(String[] args) {
        List<Producte> llistaProductes = new ArrayList<>();
        int comptador = 0;

        llistaProductes.add(new
Producte("1076543", "Zeus", "Aspirador", 180.11));
        llistaProductes.add(new
Producte("3756354", "WhitePose", "Màquina de rentar", 178.97));
        llistaProductes.add(new
Producte("1234567", "Gorth", "Nevera", 151.98));
        llistaProductes.add(new
Producte("7876161", "Gorth", "Assecadora", 159.99));

        System.out.println("TOTS ELS PRODUCTES");
        for (Producte pr : llistaProductes) {
            System.out.println(pr);
        }

        System.out.println();
        System.out.println("PRODUCTES DE PREU INFERIOR A 170€");
        for (Producte pr : llistaProductes) {
            if (pr.getPreuUnitari() < 170) {
                System.out.println(pr);
                comptador++;
            }
        }

        System.out.println();
        System.out.println("Hi ha " + comptador + " productes de cost
inferior a 170€");
    }
}
```

TOTS ELS PRODUCTES
1076543 Zeus Aspirador 180.11
3756354 WhitePose Màquina de rentar 178.97
1234567 Gorth Nevera 151.98
7876161 Gorth Assecadora 159.99

PRODUCTES DE PREU INFERIOR A 170€
1234567 Gorth Nevera 151.98

```
7876161 Gorth Assecadora 159.99
```

```
Hi ha 2 productes de cost inferior a 170€
```

53.1.2. Exemple - Cerca de productes versió Streams

```
import java.util.ArrayList;
import java.util.List;

public class AmbStreams {

    public static void main(String[] args) {
        List<Producte> llistaProductes = new ArrayList<>();

        llistaProductes.add(new
Producte("1076543", "Zeus", "Aspirador", 180.11));
        llistaProductes.add(new
Producte("3756354", "WhitePose", "Màquina de rentar", 178.97));
        llistaProductes.add(new
Producte("1234567", "Gorth", "Nevera", 151.98));
        llistaProductes.add(new
Producte("7876161", "Gorth", "Assecadora", 159.99));

        System.out.println("TOTS ELS PRODUCTES");
        llistaProductes.stream().forEach(pr ->
System.out.println(pr)); ①

        System.out.println();
        System.out.println("PRODUCTES DE PREU INFERIOR A 170€");
        llistaProductes.stream().filter(pr -> pr.getPreuUnitari()
< 170).forEach(pr -> System.out.println(pr)); ②

        System.out.println();
        long comptador = llistaProductes.stream().filter(pr ->
pr.getPreuUnitari() < 170).count(); ③
        System.out.println("Hi ha " + comptador + " productes de cost
inferior a 170€");
    }
}
```

- ① En primer lloc creem un flux amb el mètode **stream** proporcionat per la interfície **Collection** i implementat per la classe **ArrayList**. Com que només

volem mostrar tota la llista no cal que realitzem operacions addicionals, procedim directament amb la finalització del stream amb el mètode **forEach**.

Opcionalment podríem haver utilitzat la notació:

```
llistaProductes.stream().forEach(System.out::println);
```

- ❷ En aquest cas utilitzem una operació intermediària, **filter**, que retorna un altre stream, només amb els elements que compleixen la condició del filtre i que el finalitzem amb **forEach**.
- ❸ El mètode **count**, que és un mètode de finalització, retorna un **long** indicant la quantitat d'elements del fluxe resultant.

53.2. Creació de fluxos

A continuació es mostren algunes maneres de crear fluxos.

53.2.1. Directament a partir dels seus elements

```
public class StreamsTest {

    public static void main(String[] args) {
        Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9);
        stream.forEach(p -> System.out.println(p));
    }
}
```

53.2.2. A partir d'una matriu

```
public class StreamsTest {

    public static void main(String[] args) {
        Stream<Integer> stream = Stream.of(new Integer[]
{1, 2, 3, 4, 5, 6, 7, 8, 9});
        stream.forEach(p -> System.out.println(p));
    }
}
```

53.2.3. A partir d'una List

```
public class StreamsTest {

    public static void main(String[] args) {
        List<Integer> list = new ArrayList<Integer>();

        for (int i = 1; i < 10; i++) {
            list.add(i);
        }

        Stream<Integer> stream = list.stream();
        stream.forEach(p -> System.out.println(p));
    }
}
```

53.2.4. A partir d'una funció generadora

```
public class StreamsTest {

    public static void main(String[] args) {
        Stream<Integer> randomNumbers = Stream.generate(() -> (new
Random()).nextInt(100));

        randomNumbers.limit(20).forEach(System.out::println); ❶
    }
}
```

- ❶ L'operador :: és una drecera utilitzada en expressions Lambda que criden a un mètode determinat. L'operador retorna una interfície funcional.

Alguns exemples de la sintaxi:

Mètode estàtic.

```
// Lambda expression
(args) -> ClassName.staticMethodName(args)

// Method Reference
ClassName::staticMethodName
```

Mètode d'una instància.

```
// Lambda expression
(args) -> object.instanceMethodName(args)

// Method Reference
object::instanceMethodName
```

Constructor.

```
ClassName::new
```

Mètode de la classe pare.

```
super::parentMethodName
```

53.3. Convertir un Stream a una col·lecció

Després de realitzar les operacions intermèdies en els elements del flux, podem tornar a recollir els elements processats en una col·lecció mitjançant un mètode terminal.

53.3.1. Recuperar els elements en una List

```
public class StreamsTest {

    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();

        for (int i = 1; i < 10; i++) {
            list.add(i);
        }

        Stream<Integer> stream = list.stream();
        List<Integer> evenNumbersList = stream.filter(i -> i % 2 == 0)
            .collect(Collectors.toList());
        System.out.print(evenNumbersList);
    }
}
```

53.3.2. Recuperar els elements en una matriu

```
public class StreamsTest {

    public static void main(String[] args) {
        List<Integer> list = new ArrayList<Integer>();

        for (int i = 1; i < 10; i++) {
            list.add(i);
        }

        Stream<Integer> stream = list.stream();
        Integer[] evenNumbersArr = stream.filter(i -> i % 2
== 0).toArray(Integer[]::new);
        System.out.print(evenNumbersArr);
    }
}
```



Existeixen altres maneres de recollir el flux en un conjunt, un **Set**, en un **Map**, etc...

53.4. Operacions intermèdies

Les operacions intermèdies retornen el mateix flux per poder encadenar diverses crides de mètodes seguits.

A continuació és mostren els més rellevants.

53.4.1. Stream.filter()

El mètode **filter()** accepta un predicat per filtrar tots els elements del flux. Aquesta operació és intermèdia, cosa que ens permet cridar a una altra operació de flux (per exemple, **forEach()**) sobre el resultat.

```
public class StreamsTest {

    public static void main(String[] args) {
        List<String> memberNames = new ArrayList<>();
        memberNames.add("Anna");
        memberNames.add("Sergi");
    }
}
```

```

memberNames.add("Andreu");
memberNames.add("Ramon");
memberNames.add("Sara");
memberNames.add("Saúl");
memberNames.add("Yvana");
memberNames.add("Lluís");

memberNames.stream().filter((s) ->
s.startsWith("A")).forEach(System.out::println);
}
}

```

Anna
Andreu

53.4.2. Stream.map()

L'operació intermèdia **map()** converteix cada element del flux en un altre objecte mitjançant la funció donada.

L'exemple següent converteix cada cadena en una cadena **majúscula**. Però també podem utilitzar **map()** per transformar un objecte en un altre tipus.

```

public static void main(String[] args) {
    List<String> memberNames = new ArrayList<>();
    memberNames.add("Anna");
    memberNames.add("Sergi");
    memberNames.add("Andreu");
    memberNames.add("Ramon");
    memberNames.add("Sara");
    memberNames.add("Saúl");
    memberNames.add("Yvana");
    memberNames.add("Lluís");

    memberNames.stream().filter((s) -> s.startsWith("A"))
        .map(String::toUpperCase)
        .forEach(System.out::println);
}

```

ANNA
ANDREU

53.4.3. Stream.sorted()

El mètode **sorted()** és una operació intermèdia que retorna una vista ordenada del flux. Els elements del flux s'ordenen per ordre natural tret que passem un **Comparator** personalitzat.

```
public class StreamsTest {

    public static void main(String[] args) {
        List<String> memberNames = new ArrayList<>();
        memberNames.add("Anna");
        memberNames.add("Sergi");
        memberNames.add("Andreu");
        memberNames.add("Ramon");
        memberNames.add("Sara");
        memberNames.add("Saúl");
        memberNames.add("Yvana");
        memberNames.add("Lluís");

        memberNames.stream().sorted()
            .map(String::toUpperCase)
            .forEach(System.out::println);
    }
}
```

ANDREU
ANNA
LLUÍS
RAMON
SARA
SAÚL
SERGI
YVANA



Tingueu en compte que en l'exemple anterior, l'ordre de la col·lecció origen no es modifica!!

53.5. Operacions terminals

Les operacions de terminals retornen un resultat d'un tipus determinat després de processar tots els elements del flux.

Quan s'invoca l'operació terminal en un flux, s'iniciarà la iteració del flux i de qualsevol dels fluxos encadenats. Un cop feta la iteració, es torna el resultat de l'operació terminal.

53.5.1. Stream.forEach()

El mètode **forEach()** ajuda a iterar tots els elements d'un flux i a realitzar alguna operació en cadascun d'ells. L'operació a realitzar es passa com a expressió lambda.

```
memberNames.forEach(System.out::println);
```

53.5.2. Stream.collect()

El mètode **collect()** s'utilitza per rebre elements d'un flux i emmagatzemar-los en una col·lecció.

```
public class StreamsTest {

    public static void main(String[] args) {
        List<String> memberNames = new ArrayList<>();
        memberNames.add("Anna");
        memberNames.add("Sergi");
        memberNames.add("Andreu");
        memberNames.add("Ramon");
        memberNames.add("Sara");
        memberNames.add("Saúl");
        memberNames.add("Yvana");
        memberNames.add("Lluís");

        List<String> memNamesInUppercase = memberNames.stream().sorted()
            .map(String::toUpperCase)
            .collect(Collectors.toList());

        System.out.print(memNamesInUppercase);
    }
}
```

```
[ANDREU, ANNA, LLUÍS, RAMON, SARA, SAÚL, SERGI, YVANA]
```

53.5.3. Stream.match()

Es poden utilitzar diverses operacions **match** per comprovar si un determinat predicat coincideix amb els elements del flux. Totes aquestes operacions de coincidència són terminals i retornen un resultat booleà.

```
public class StreamsTest {

    public static void main(String[] args) {
        List<String> memberNames = new ArrayList<>();
        memberNames.add("Anna");
        memberNames.add("Sergi");
        memberNames.add("Andreu");
        memberNames.add("Ramon");
        memberNames.add("Sara");
        memberNames.add("Saúl");
        memberNames.add("Yvana");
        memberNames.add("Lluís");

        boolean matchedResult = memberNames.stream()
            .anyMatch((s) -> s.startsWith("A"));

        System.out.println(matchedResult);

        matchedResult = memberNames.stream()
            .allMatch((s) -> s.startsWith("A"));

        System.out.println(matchedResult);

        matchedResult = memberNames.stream()
            .noneMatch((s) -> s.startsWith("A"));

        System.out.println(matchedResult);
    }
}
```

```
true
false
false
```

53.5.4. Stream.count()

El mètode **count()** és una operació de terminal que retorna el nombre d'elements del flux com a **long**.

```
public class StreamsTest {  
  
    public static void main(String[] args) {  
        List<String> memberNames = new ArrayList<>();  
        memberNames.add("Anna");  
        memberNames.add("Sergi");  
        memberNames.add("Andreu");  
        memberNames.add("Ramon");  
        memberNames.add("Sara");  
        memberNames.add("Saúl");  
        memberNames.add("Yvana");  
        memberNames.add("Lluís");  
  
        long totalMatched = memberNames.stream()  
            .filter(s -> s.startsWith("A"))  
            .count();  
  
        System.out.println(totalMatched);  
    }  
}
```

2

53.5.5. Stream.reduce()

El mètode **reduce()** realitza una reducció dels elements del flux amb la funció donada. El resultat és un **Optional** que manté el valor reduït.

En l'exemple donat, reduïm totes les cadenes concatenant-les mitjançant un separador #.

Classe Optional

Generalment, els dissenyadors d'API posen els documents descriptius de java a les API i hi mencionen que l'API pot retornar un valor **null** i en quins casos. El problema és que és possible que la persona que crida a l'API no hagi llegit la documentació associada i s'hagi oblidat de gestionar el cas **null** cosa que serà un error en el futur.

Per evitar el problema anterior s'ha dissenyat la classe **Optional** que indica al programador que el valor que encapsula pot ser **null** i a més evita l'assignació d'aquesta referència.

De la documentació de Java:

```
public final class Optional<T> extends Object
```

Un **Optional** és un objecte contenidor que **pot contenir o no un valor no null**. Si hi ha un valor, **isPresent()** retorna **true**. Si no hi ha cap valor, l'objecte es considera buit i **isPresent()** retorna **false**.

53.6. Operacions interruptibles

Tot i que les operacions de transmissió es realitzen a tots els elements d'una col·lecció que satisfan un predicat, sovint es vol interrompre l'operació en el moment que es troba un element coincident durant la iteració.

En la iteració externa, ho farem amb el bloc if-else. A les iteracions internes, com ara als fluxos, hi ha alguns mètodes que podem utilitzar per a aquest propòsit.

53.6.1. Stream.anyMatch ()

El mètode **anyMatch()** tornarà **true** quan es compleixi una condició passada com a predicat. Un cop trobat un valor coincident, no es processaran més elements del flux.

En l'exemple donat, tan aviat com es troba una cadena que comença amb la lletra "A", la transmissió finalitzarà i es retornarà el resultat.

```
public class StreamsTest {

    public static void main(String[] args) {
        List<String> memberNames = new ArrayList<>();
        memberNames.add("Anna");
        memberNames.add("Sergi");
        memberNames.add("Andreu");
        memberNames.add("Ramon");
        memberNames.add("Sara");
        memberNames.add("Saúl");
        memberNames.add("Yvana");
        memberNames.add("Lluís");

        boolean matched = memberNames.stream()
            .anyMatch((s) -> s.startsWith("A"));

        System.out.println(matched);
    }
}
```

true

53.6.2. Stream.findFirst()

El mètode **findFirst()** retornarà el primer element del flux i després no processarà més elements.

```
public class StreamsTest {

    public static void main(String[] args) {
        List<String> memberNames = new ArrayList<>();
        memberNames.add("Anna");
        memberNames.add("Sergi");
        memberNames.add("Andreu");
        memberNames.add("Ramon");
        memberNames.add("Sara");
        memberNames.add("Saúl");
        memberNames.add("Yvana");
        memberNames.add("Lluís");

        String firstMatchedName = memberNames.stream()
            .filter((s) -> s.startsWith("L"))
```

```
        .findFirst()
        .get();

    System.out.println(firstMatchedName);
}

}
```

Lluís

53.7. Streams paral·lels

En qualsevol dels exemples esmentats anteriorment, si volem fer una feina particular utilitzant diversos fils d'execució en "cores" paral·lels, tot el que hem de fer és cridar al mètode **parallelStream()** en lloc del mètode **stream()**.

54

JavaFX - Esdeveniments

54.1. Esdeveniments

Per respondre a un esdeveniment cal escriure codi per processar l'acció que ha desencadenat l'esdeveniment, un click de ratolí , passar el ratolí per damunt d'un node, un a pulsació de tecla, etc...

A tal efecte existeix una interfície funcional anomenada **EventHandler** amb la següent definició:

```
@FunctionalInterface  
public interface EventHandler<T extends Event> extends EventListener {  
    void handle(T event);  
}
```

on la interfície **EventListener** és una interfície de "marcat" definida com:

```
public interface EventListener {  
}
```

En Java, caldrà crear un objecte capaç de gestionar l'esdeveniment, aquest objecte s'anomena **EventHandler**.

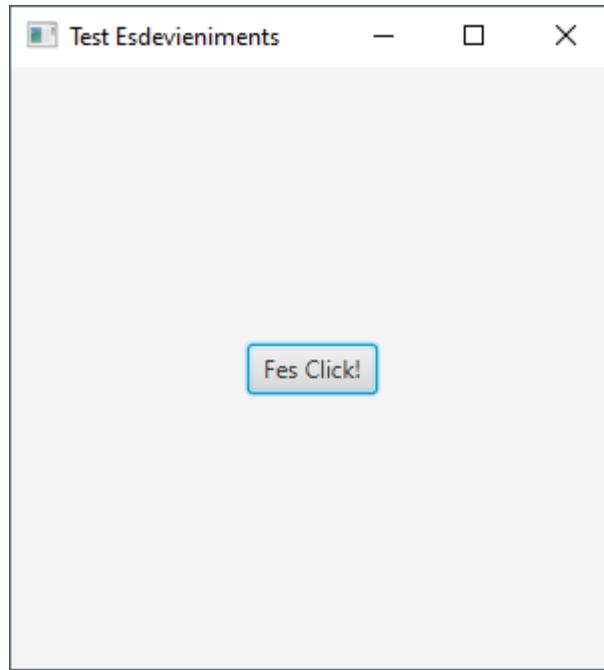
Per construir un objecte de tipus **EventHandler** calen dues coses:

1. L'objecte ha de ser una instància de la interfície **EventHandler<T extends Event>** on
 - a. T és un subtípus de **Event**.

- b. **Event** és el tipus d'esdeveniment que s'ha produït, per exemple un **ActionEvent** representa un click de ratolí mentre que un **MouseEvent** representa un moviment de ratolí.
2. Un cop tenim l'objecte que respondrà a l'esdeveniment caldrà associar aquest objecte amb l'objecte **origen de l'esdeveniment** utilitzant alguns dels mètodes **setOn.....**. Per exemple **source.setOnAction()** o **source, setOnMouseEvent()**.

Vegem-ho amb un exemple, tenim la següent aplicació:

```
public class TestEsdeveniments extends Application {  
  
    @Override  
    public void start(Stage primaryStage) {  
        StackPane pane = new StackPane();  
  
        Button btn = new Button("Fes Click!");  
        pane.getChildren().add(btn);  
  
        Scene scene = new Scene(pane, 300, 300);  
  
        primaryStage.setTitle("Test Esdevieniments");  
        primaryStage.setScene(scene);  
        primaryStage.show();  
    }  
  
    public static void main(String[] args) {  
        launch(args);  
    }  
}
```



Volem gestionar l'esdeveniment de click del botó, es tracta d'un esdeveniment de tipus **ActionEvent**, per tant creem una classe derivada de **EventHandler<ActionEvent>** i implementem el comportament de resposta a l'esdeveniment.

Tenim quatre maneres diferents de tractar el problema anterior:

54.1.1. Creant una classe que implementa EventHandler<ActionEvent>

```
class ClickHandlerClass implements EventHandler<ActionEvent> {

    @Override
    public void handle(ActionEvent event) {
        System.out.println("Has fet click!!!");
    }
}
```

Finalment enllacem un objecte de la classe anterior amb el node botó.

```

public class TestEsdeveniments extends Application {

    @Override
    public void start(Stage primaryStage) {
        StackPane pane = new StackPane();

        Button btn = new Button("Fes Click!");
        btn.setOnAction(new ClickHandlerClass()); ①
        pane.getChildren().add(btn);

        Scene scene = new Scene(pane, 300, 300);

        primaryStage.setTitle("Test Esdeveniments");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}

```

54.1.2. Utilitzant una classe anònima

Gestió d'un esdeveniment amb una inner class.

```

public class TestJavaFX extends Application {

    @Override
    public void start(Stage primaryStage) {
        StackPane pane = new StackPane();

        Button btn = new Button("Fes Click!");
        btn.setOnAction(new EventHandler<ActionEvent>() {①
            @Override
            public void handle(ActionEvent event) {
                System.out.println("Has fet click!!!");
            }
        });
        pane.getChildren().add(btn);

        Scene scene = new Scene(pane, 300, 300);
    }
}

```

```

        primaryStage.setTitle("Test Esdevieniments");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}

```

54.1.3. Utilitzant una expressió lambda

Gestió d'un esdeveniment amb una expressió Lambda.

```

public class TestJavaFX extends Application {

    @Override
    public void start(Stage primaryStage) {
        StackPane pane = new StackPane();

        Button btn = new Button("Fes Click!");
        btn.setOnAction((ActionEvent event) -> { ❶
            System.out.println("Has fet click!!!");
        });
        pane.getChildren().add(btn);

        Scene scene = new Scene(pane, 300, 300);

        primaryStage.setTitle("Test Esdevieniments");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}

```

54.1.4. Centralitzant la gestió d'esdeveniments en un sol mètode

```
public class TestJavaFX extends Application implements  
EventHandler<ActionEvent> { ①  
  
    private Button btn1; ②  
    private Button btn2;  
  
    @Override  
    public void start(Stage primaryStage) {  
        StackPane pane = new StackPane();  
  
        btn1 = new Button("Botó 1");  
        btn1.setOnAction(this); ③  
        btn2 = new Button("Botó");  
        btn2.setOnAction(this); ④  
        pane.getChildren().addAll(btn1, btn2);  
  
        Scene scene = new Scene(pane, 300, 300);  
  
        primaryStage.setTitle("Test Esdeveniments");  
        primaryStage.setScene(scene);  
        primaryStage.show();  
    }  
  
    public static void main(String[] args) {  
        launch(args);  
    }  
  
    @Override  
    public void handle(ActionEvent event) { ⑤  
        if (event.getSource() == btn1) { ⑥  
            System.out.println("Has fet click al botó 1");  
        } else if (event.getSource() == btn2) {  
            System.out.println("Has fet click al botó 2");  
        }  
    }  
}
```

- ⑤ La pròpia classe implementa **EventHandler<ActionEvent>** i per tant ha d'implementar el mètode **handle**.
- ⑥ Podem comprovar l'origen de l'esdeveniment amb el mètode **getSource()** de **event**, en funció d'aquest origen implementem la resposta a l'esdeveniment.
- ⑦ Cal posar la pròpia classe **TestJavaFX** com a observer del botó, d'aquesta manera cada cop que el botó dispara un **ActionEvent** es notificarà cridant al mètode **handle** de la classe.

54.2. Alguns esdeveniments comuns

54.2.1. Esdeveniments del ratolí

Els esdeveniments de tipus **MouseEvent** es disparen quan es prem, s'allibera o es fa clic a un botó del ratolí, quan es mou o s'arrossega el ratolí sobre un **Node** o una **Scene**.

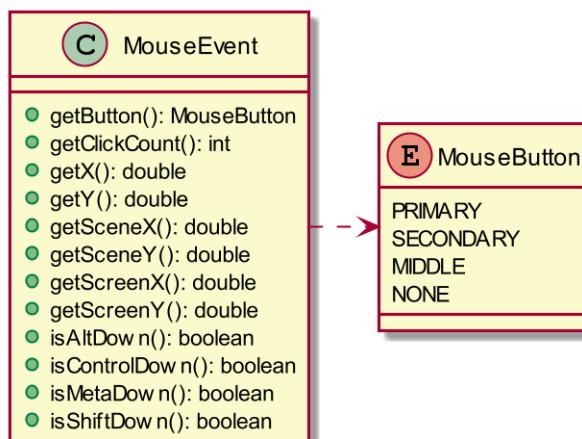


Figura 54.1. `javafx.scene.input.MouseEvent`

```

package testjavafx;

import javafx.application.Application;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.Pane;
import javafx.scene.text.Text;
  
```

```

import javafx.stage.Stage;

public class TestEsdeveniments extends Application {

    @Override
    public void start(Stage primaryStage) {
        Pane pane = new Pane();
        Text text = new Text(20, 20, "Abracadabra");
        pane.getChildren().addAll(text);
        text.setOnMouseDragged(new EventHandler<MouseEvent>() {
            @Override
            public void handle(MouseEvent event) {
                text.setX(event.getX());
                text.setY(event.getY());
            }
        });

        Scene scene = new Scene(pane, 300, 100);
        primaryStage.setTitle("MouseEvent Test");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}

```

54.2.2. Esdeveniments del teclat

L'esdeveniment **KeyEvent** es dispara quan es prem s'allibera o s'escriu una tecla en un **Node** o en una **Scene**.

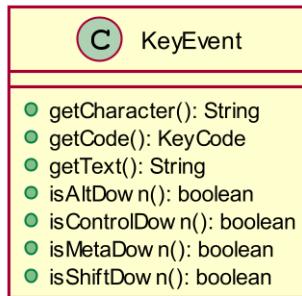


Figura 54.2. javafx.scene.input.KeyEvent

Cada esdeveniment de teclat té associat un codi que s'obté cridant al mètode **getCode()** del **KeyEvent**. Aquests codis són constants definides dins del la classe **KeyCode**

```
package testjavafx;

import javafx.application.Application;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.input.KeyEvent;
import javafx.scene.layout.Pane;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class TestEsdeveniments extends Application {

    @Override
    public void start(Stage primaryStage) {

        Pane pane = new Pane();
        Text text = new Text(20, 20, "A");

        pane.getChildren().add(text);
        text.setOnKeyPressed(new EventHandler<KeyEvent>() {
            @Override
            public void handle(KeyEvent event) {
                switch (event.getCode()) {
                    case DOWN:
                        text.setY(text.getY() + 10);
                        break;
                    case UP:
                        text.setY(text.getY() - 10);
                        break;
                    case LEFT:
                        text.setX(text.getX() - 10);
                        break;
                    case RIGHT:
                        text.setX(text.getX() + 10);
                        break;
                    default:
                        if
                            (Character.isLetterOrDigit(event.getText().charAt(0))) {
                                text.setText(event.getText());
                            }
                }
            }
        });
    }
}
```

```
        }
    }
});

Scene scene = new Scene(pane);
primaryStage.setTitle("KeyEvent Test");
primaryStage.setScene(scene);
primaryStage.show();

text.requestFocus(); // Volem rebre els esdeveniments de teclat
a text
}

public static void main(String[] args) {
    launch(args);
}
}
```

Patró Model-vista-Controlador

L'arquitectura **Model–Vista–Controlador** (MVC) és un patró de disseny utilitzat per a la implementació d'interfícies d'usuari. Aquest patró divideix l'aplicació en tres parts interconnectades: el **model de dades**, la **interfície usuari** i la **lògica de control**.

Es poden definir els components de MVC de la següent forma:

El Model

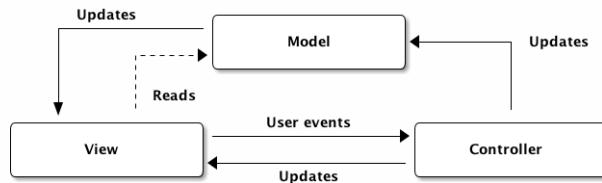
és la representació de la informació amb la qual el sistema opera, per tant gestiona tots els accses a aquesta informació, tant consultes com actualitzacions, implementant també els privilegis d'accés que s'hagin descrit en les especificacions de l'aplicació (lògica del negoci). Envia a la **vista** aquella part de la informació que en cada moment li sol·licita perquè sigui mostrada (típicament a un usuari). Les peticions d'accés o manipulació d'informació arriben al **model** a través del **controlador**.

El Controlador

respon a esdeveniments (usualment accions de l'usuari) i invoca peticions al **model** quan es fa alguna sol·licitud sobre la informació (per exemple, editar un document o un registre en una base de dades). També pot enviar comandes a la **vista** associada si es sol·licita un canvi en la forma en què es presenta el **model'** (per exemple, desplaçament o scroll per un document o per diferents registres d'una base de dades), per tant es podria dir que el **controlador** fa d'intermediari entre la **vista** i el **model**.

La Vista

presenta el **model** (informació i lògica de negoci) en un format adequat per a interaccionar (usualment la interfície d'usuari) per tant cal que proveeixi la informació de sortida del **model**.



55.1. Avantatges i problemes del model

55.1.1. Avantatges

Desenvolupament simultani

Més d'un desenvolupador pot treballar simultàniament sobre el model, el controlador o les vistes.

Alta cohesió

El patró MVC afavoreix l'agrupament lòtic de les accions relacionades dins d'un controlador. Les vistes per un model específic també estan agrupades.

Cohesió

La cohesió, en termes de Software, fa referència a la mesura de quant un mòdul d'un sistema té una sola responsabilitat. Un mòdul o classe amb alta cohesió serà aquell que mantingui una elevada relació entre les seves funcionalitats mantenint l'enfoc en un únic propòsit. Aquest concepte està relacionat amb el principi de responsabilitat única.

Baix acoblament

La pròpia naturalesa del patró MVC propicia el baix acoblament entre models, vistes i controladors.

Acoblament

L'acoblament, en termes de Software, fa referència a la relació que guarden entre ells els mòduls o les classes d'un sistema. Un baix acoblament dins d'un sistema indica que els mòduls o les classes no coneixen o coneixen molt poc del funcionament intern dels altres mòduls o classes, evitant una dependència forta entre ells. Aquest concepte està relacionat amb el principi d'obert-tancat.

Facilitat de modificació

La separació de responsabilitats promoguda pel model MVC facilita el futur manteniment i modificació del sistema.

Múltiples vistes per un model

El patró permet que els models tinguin més d'una vista amb facilitat.

55.1.2. Problemes

Navegació del codi

La navegació al codi pot esdevenir complexa degut a les capes d'indirecció afegides pel propi model.

Excessiu codi repetitiu

Al dividir el codi en tres parts es genera molt de codi que només serveix per satisfer el patró.

55.2. Esquelet del patró

A continuació es mostra una proposta de l'esquelet del patró MVC en JavaFX i a continuació una implementació concreta.

55.2.1. El model

El model és la classe menys definible de totes, depèn exclusivament de l'aplicació, en general es tracta d'una classe que proporciona el contingut de a les dades que es mostraran a la vista.

En funció del tipus d'aplicació les dades proporcionades pel model poden ser tant tant de lectura com d'escriptura.

```
package mvc.models;

public class Model {

    public Model() {
        // TODO: Implementació específica de cada model
    }

}
```

55.2.2. *La vista*

La vista és la responsable de "dibuixar" el formulari i els controls que conté.

```
package mvc.views;

import javafx.scene.layout.Pane;

public class View {

    private Pane rootPane;
    // TODO: Crear un atribut privat per a cada control

    public View() {
        // TODO: Inicialització dels diferents controls
    }

    public Pane getRootPane() {
        return rootPane;
    }

    // TODO: Implementar getters per a tots els controls accessibles
}
```

55.2.3. *El controlador*

El controlador té la responsabilitat d'enllaçar les dades proporcionades pel model als controls proporcionats per la vista. Aquest enllaç pot ser de només lectura o de lectura/escriptura.

```
package mvc.controllers;

import mvc.models.Model;
import mvc.views.View;

public class Controller {

    private Model model;
    private View view;

    public Controller(Model m, View v) {
        model = m;
        view = v;
        initView();
    }

    public void initView() {
        // TODO: Gestionar el contingut inicial dels Controls de la
        vista
    }

    public void initController() {
        // TODO: Implementar totes les classes EventHandler
    }

}
```

55.2.4. El mètode main

El punt d'entrada a una aplicació JavaFX dissenyada amb MVC es pot implementar com s'indica a continuació.

```
package mvc;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;
import mvc.controllers.Controller;
import mvc.models.Model;
import mvc.views.View;

public class Mvc extends Application {
```

```

@Override
public void start(Stage primaryStage) {
    Model m = new Model();
    View v = new View();
    Controller c = new Controller(m, v);
    c.initController();

    primaryStage.setTitle("MVC");
    primaryStage.setScene(new Scene(v.getRootPane(), 300, 200));
    primaryStage.show();
}

public static void main(String[] args) {
    launch(args);
}

}

```

55.3. Un exemple d'implementació

I a continuació és mostra un exemple concret d'implementació del patró mostrat prèviament.

55.3.1. El model

Classe Model.

```

package mvc.models;

public class Model {

    private String nom;
    private String primerCognom;
    private String segonCognom;

    public Model(String nom, String primerCognom, String segonCognom) {
        this.nom = nom;
        this.primerCognom = primerCognom;
        this.segonCognom = segonCognom;
    }

    public String getNom() {
        return nom;
    }
}

```

```
public void setNom(String nom) {
    this.nom = nom;
}

public String getPrimerCognom() {
    return primerCognom;
}

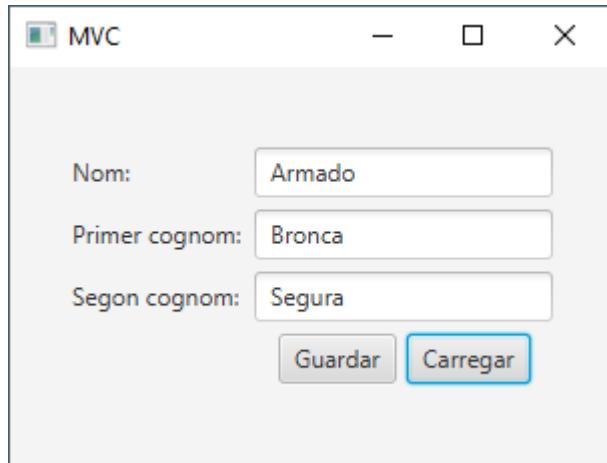
public void setPrimerCognom(String primerCognom) {
    this.primerCognom = primerCognom;
}

public String getSegonCognom() {
    return segonCognom;
}

public void setSegonCognom(String segonCognom) {
    this.segonCognom = segonCognom;
}

}
```

55.3.2. La vista



Classe View.

```
package mvc.views;
```

```
import javafx.geometry.HPos;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;

public class View {

    private GridPane rootPane;
    private Label nomLabel;
    private Label primerCognomLabel;
    private Label segonCognomLabel;
    private TextField nomTextField;
    private TextField primerCognomTextField;
    private TextField segonCognomTextField;
    private HBox buttonPane;
    private Button guardarButton;
    private Button carregarButton;

    public View() {
        rootPane = new GridPane();
        rootPane.setAlignment(Pos.CENTER);
        rootPane.setPadding(new Insets(11.5, 12.5, 13.5, 14.5));
        rootPane.setHgap(5.5);
        rootPane.setVgap(5.5);

        nomLabel = new Label("Nom:");
        primerCognomLabel = new Label("Primer cognom:");
        segonCognomLabel = new Label("Segon cognom:");
        nomTextField = new TextField();
        primerCognomTextField = new TextField();
        segonCognomTextField = new TextField();
        buttonPane = new HBox();
        buttonPane.setSpacing(5);
        guardarButton = new Button("Guardar");
        carregarButton = new Button("Carregar");

        rootPane.add(nomLabel, 0, 0);
        rootPane.add(primerCognomLabel, 0, 1);
        rootPane.add(segonCognomLabel, 0, 2);
        rootPane.add(nomTextField, 1, 0);
        rootPane.add(primerCognomTextField, 1, 1);
```

```
rootPane.add(segonCognomTextField, 1, 2);

buttonPane.getChildren().addAll(guardarButton, carregarButton);
rootPane.add(buttonPane, 1, 3);
buttonPane.setAlignment(Pos.CENTER);
GridPane.setHalignment(buttonPane, HPos.RIGHT);
}

public GridPane getRootPane() {
    return rootPane;
}

public Label getNomLabel() {
    return nomLabel;
}

public Label getPrimerCognomLabel() {
    return primerCognomLabel;
}

public Label getSegonCognomLabel() {
    return segonCognomLabel;
}

public TextField getNomTextField() {
    return nomTextField;
}

public TextField getPrimerCognomTextField() {
    return primerCognomTextField;
}

public TextField getSegonCognomTextField() {
    return segonCognomTextField;
}

public HBox getButtonPane() {
    return buttonPane;
}

public Button getGuardarButton() {
    return guardarButton;
}

public Button getCarregarButton() {
```

```
        return carregarButton;
    }
}
```

55.3.3. El Controlador

Classe Controller.

```
package mvc.controllers;

import mvc.models.Model;
import mvc.views.View;

import javafx.event.ActionEvent;
import javafx.event.EventHandler;

public class Controller {

    private Model model;
    private View view;

    public Controller(Model m, View v) {
        model = m;
        view = v;
        initView();
    }

    public void initView() {
        carregarRegistre();
    }

    public void initController() {
        view.getCarregarButton().setOnAction(new
EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                carregarRegistre();
            }
        });

        view.getGuardarButton().setOnAction(new
EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
```

```

        guardarRegistre();
    }
});

private void carregarRegistre() {
    view.getNomTextField().setText(model.getNom());

    view.getPrimerCognomTextField().setText(model.getPrimerCognom());
    view.getSegonCognomTextField().setText(model.getSegonCognom());
}

private void guardarRegistre() {
    model.setNom(view.getNomTextField().getText());

    model.setPrimerCognom(view.getPrimerCognomTextField().getText());
    model.setSegonCognom(view.getSegonCognomTextField().getText());
}
}

```

55.3.4. El mètode main

Classe Main.

```

package mvc;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;
import mvc.controllers.Controller;
import mvc.models.Model;
import mvc.views.View;

public class Mvc extends Application {

    @Override
    public void start(Stage primaryStage) {
        Model m = new Model("Armando", "Bronca", "Segura");
        View v = new View();
        Controller c = new Controller(m, v);
        c.initController();

        primaryStage.setTitle("MVC");
        primaryStage.setScene(new Scene(v.getRootPane(), 300, 200));
        primaryStage.show();
    }
}

```

```
}
```

```
public static void main(String[] args) {
```

```
    launch(args);
```

```
}
```

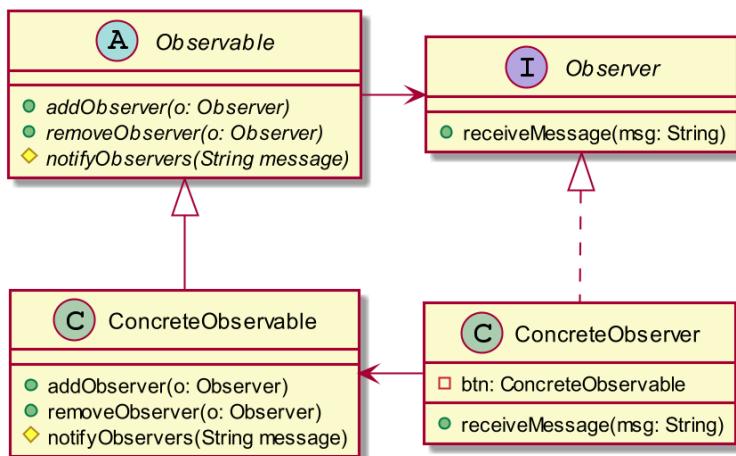
```
}
```

56

Patró Observer

El patró **Observer** s'utilitza quan volem definir una dependència d'un a molts objectes de tal manera que quan l'objecte canvia l'estat, tots els objectes dependents siguin notificats automàticament.

56.1. Definició del patró



56.2. Implementació en Java

Classe Observer.

```
package observerpattern;

public interface Observer {
    void receiveMessage(String message);
```

```
}
```

Classe Observable.

```
package observerpattern;

public abstract class Observable {
    abstract public void addObserver(Observer o);
    abstract public void removeObserver(Observer o);
    abstract protected void notifyObservers(String message);
}
```

Classe ConcreteObserver.

```
package observerpattern;

public class ConcreteObserver implements Observer {

    private ConcreteObservable observable;

    public ConcreteObserver() {
        observable = new ConcreteObservable();
        observable.addObserver(this);
    }

    public void run() {
        observable.run();
    }

    @Override
    public void receiveMessage(String message) {
        System.out.println("Concrete Observer: " + message);
    }
}
```

Classe ConcreteObservable.

```
package observerpattern;

import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

public class ConcreteObservable extends Observable {
```

```
private Scanner in;
private List<Observer> observers;

public ConcreteObservable() {
    in = new Scanner(System.in);
    observers = new ArrayList<>();
}

public void run() {

    while (true) {
        String valor = in.next();
        notifyObservers(valor);
    }
}

@Override
public void addObserver(Observer o) {
    observers.add(o);
}

@Override
public void removeObserver(Observer o) {
    observers.remove(o);
}

@Override
protected void notifyObservers(String message) {
    for (Observer o : observers) {
        o.receiveMessage(message);
    }
}
}
```

Classe Main.

```
package observerpattern;

public class Main {

    public static void main(String[] args) {
        testInnerClass();
        testConcreteObserver();
    }
}
```

```
public static void testInnerClass() {
    Observer o1 = new Observer() {
        @Override
        public void receiveMessage(String message) {
            if (message.equals("message")) {
                System.out.println("o1: message received!!");
            }
        }
    };

    Observer o2 = new Observer() {
        @Override
        public void receiveMessage(String message) {
            if (message.equals("message")) {
                System.out.println("o2: message received!!");
            }
        }
    };
    ConcreteObservable observable = new ConcreteObservable();
    observable.addObserver(o1);
    observable.addObserver(o2);
    observable.run();
}

public static void testConcreteObserver() {
    ConcreteObserver f = new ConcreteObserver();
    f.run();
}
}
```

Vincular atributs de dues classes

Imaginem que tenim les següents classes:

Classe ClasseA.

```
public class ClasseA {  
    private int valor;  
  
    public int getValor() {  
        return valor;  
    }  
  
    public void setValor(int valor) {  
        this.valor = valor;  
    }  
}
```

Classe ClasseB.

```
public class ClasseB {  
    private int valor;  
  
    public int getValor() {  
        return valor;  
    }  
  
    public void setValor(int valor) {  
        this.valor = valor;  
    }  
}
```

I volem aconseguir **una manera de vincular els dos valors de les dues classes** de manera que un canvi al valor de **ClasseA** produueixi automàticament un canvi al valor de la **ClasseB**.



Per simplificar només considerarem l'enllaç en un sol sentit,
de **ClasseA** a **ClasseB**.

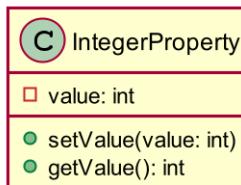
A més, voldríem una solució genèrica, és a dir, voldríem una solució que funcionés automàticament per a qualsevol atribut de tipus **int** de qualsevol classe.

Una solució seria **encapsular el tipus intr** dins d'una nova classe amb capacitat de ser **observada** utilitzant el **patró observer** i amb capacitat de ser **observadors** utilitzant també el **patró observer**.

D'aquesta manera un atribut **int** d'una classe podria estar pendent d'un altre atribut **int** d'una altra classe i quant el primer canviés canviar el segon automàticament.

Anem a implementar-ho:

1. En primer lloc volem una nova classe que encapsuli un valor **int** per posteriorment donar-li facultat d'observar altres enters i ser observat per altres enters. L'anomenarem **IntegerProperty**.



2. Un cop hem encapsulat el tipus **int** utilitzarem aquesta implementació dins de les classes **ClasseA** i **ClasseB**.

Classe ClasseA.

```
public class ClasseA {
    private IntegerProperty valor = new IntegerProperty();

    public int getValor() {
        return valor.getValue();
```

```

    }

    public void setValor(int valor) {
        this.valor.setValue(valor);
    }

    public IntegerProperty valorProperty() {
        return valor;
    }
}

```

Classe ClasseB.

```

public class ClasseB {
    private IntegerProperty valor = new IntegerProperty();

    public int getValor() {
        return valor.getValue();
    }

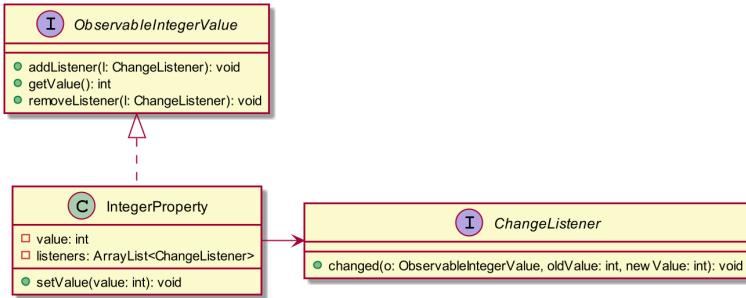
    public void setValor(int valor) {
        this.valor.setValue(valor);
    }

    public IntegerProperty valorProperty() {
        return valor;
    }
}

```

3. A continuació cal que la classe **IntegerProperty** sigui capaç d'emmagatzemar la llista de les classes que estan escoltant possibles canvis de la propietat. Per fer-ho utilitzem el patró Observer sobre **IntegerProperty**.

El mètode **setValue** notificarà a tota la llista de listeners que hi ha hagut un canvi en el valor de la propietat.



```

public class IntegerProperty implements ObservableIntegerValue {

    private int value;
    private ArrayList<ChangeListener> listeners = new ArrayList<>();

    public void setValue(int value) {
        for (ChangeListener l : listeners) {
            l.changed(this, this.value, value);
        }
        this.value = value;
    }

    @Override
    public void addListener(ChangeListener listener) {
        listeners.add(listener);
    }

    @Override
    public int getValue() {
        return value;
    }

    @Override
    public void removeListener(ChangeListener listener) {
        listeners.remove(listener);
    }

}

```

4. En aquest punt ja detectem els canvis a la propietat:

```

package bindpattern;

```

```

public class BindPattern {

    public static void main(String[] args) {
        ClasseA a = new ClasseA();
        ClasseB b = new ClasseB();

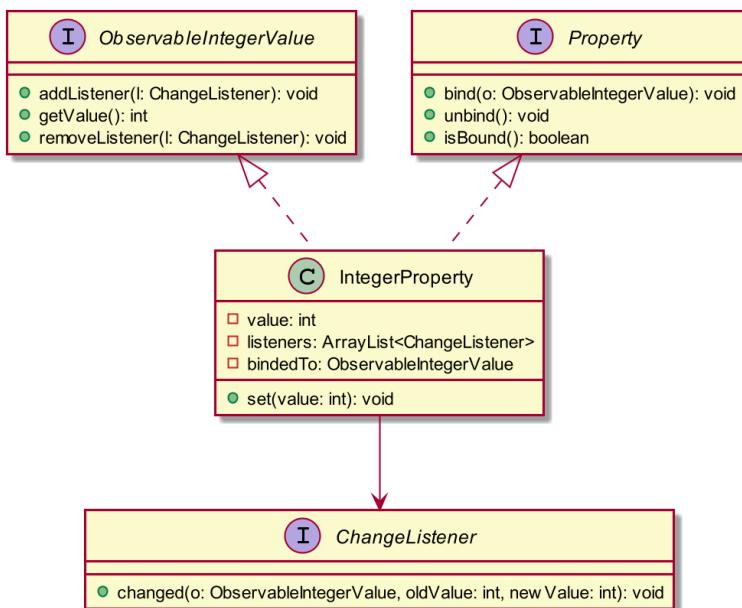
        a.valorProperty().addListener(new ChangeListener() {
            @Override
            public void changed(ObservableIntegerValue
observable, int oldValue, int newValue) {
                System.out.println("El valor ha canviat!! " +
oldValue + " -> " + newValue);
            }
        });

        a.setValor(10);
    }
}

```

Però voldríem encapsular la creació dels listeners dins de la pròpia propietat.

5. Per fer-ho, implementem un mètode **bind** que agafarà un **ObservableIntegerValue** que segurament serà un altre **IntegerProperty** i li afegirà un nou listener que en el seu mètode **changed** modificarà el **valor** propi pel nou valor passat al mètode **changed** quan es cridi al mètode **setValor** del **ObservableIntegerValue** passat per paràmetre.



```

public class IntegerProperty implements ObservableIntegerValue,
Property {

    . . .

    private ObservableIntegerValue bindedTo = null;

    . . .

    @Override
    public void bind(ObservableIntegerValue observable) {
        if (bindedTo == null) {
            observable.addListener(new ChangeListener() {
                @Override
                public void changed(ObservableIntegerValue
observable, int oldValue, int newValue) {
                    value = newValue;
                }
            });
            bindedTo = observable;
        }
    }

    @Override
  
```

```

public void unbind() {
    bindedTo = null;
}

public boolean isBound() {
    return bindedTo == null ? false : true;
}
}

```

57.1. El codi complet

```

@FunctionalInterface
public interface ChangeListener {
    void changed(ObservableIntegerValue observable, int oldValue, int
    newValue);
}

```

```

public interface Property {
    public void bind(ObservableIntegerValue o);
    public void unbind();
    public boolean isBound();
}

```

```

public interface ObservableIntegerValue {
    void addListener(ChangeListener listener);
    int getValue();
    void removeListener(ChangeListener listener);
}

```

```

public class ClasseA {
    private IntegerProperty valor = new IntegerProperty();

    public int getValor() {
        return valor.getValue();
    }

    public void setValor(int valor) {
        this.valor.setValue(valor);
    }

    public IntegerProperty valorProperty() {
}

```

```

    return valor;
}
}
```

```

public class ClasseB {
    private IntegerProperty valor = new IntegerProperty();

    public int getValor() {
        return valor.getValue();
    }

    public void setValor(int valor) {
        this.valor.setValue(valor);
    }

    public IntegerProperty valorProperty() {
        return valor;
    }
}
```

```

public class IntegerProperty implements ObservableIntegerValue, Property
{
    private int value;
    private ArrayList<ChangeListener> listeners = new ArrayList<>();
    private ObservableIntegerValue bindedTo = null;

    public void setValue(int value) {
        for (ChangeListener l : listeners) {
            l.changed(this, this.value, value);
        }
        this.value = value;
    }

    @Override
    public void bind(ObservableIntegerValue observable) {
        if (bindedTo == null) {
            observable.addListener(new ChangeListener() {
                @Override
                public void changed(ObservableIntegerValue
observable, int oldValue, int newValue) {
                    value = newValue;
                }
            });
        }
    }
}
```

```

        });
        bindedTo = observable;
    }
}

@Override
public void unbind() {
    bindedTo = null;
}

public boolean isBound() {
    return bindedTo == null ? false : true;
}

@Override
public void addListener(ChangeListener listener) {
    listeners.add(listener);
}

@Override
public int getValue() {
    return value;
}

@Override
public void removeListener(ChangeListener listener) {
    listeners.remove(listener);
}

}

```

```

public class BindPattern {

    public static void main(String[] args) {
        ClasseA a = new ClasseA();
        ClasseB b = new ClasseB();

        /*
        a.valorProperty().addListener(new ChangeListener() {
            @Override
            public void changed(ObservableIntegerValue observable, int
oldValue, int newValue) {
                System.out.println("El valor ha canviat!! " + oldValue +
" -> " + newValue);
        }
    }
}

```

```
        }
    });
/*
```

```
    a.setValor(10);
    b.setValor(50);
    a.valorProperty().bind(b.valorProperty());
    System.out.println("Valor de a: " + a.getValor());
    System.out.println("Valor de b: " + b.getValor());
    System.out.println("Modifiquem valor de b: 23");
    b.setValor(23);
    System.out.println("Valor de a: " + a.getValor());
    System.out.println("Valor de b: " + b.getValor());
}
}
```

58

JavaFX - Binding

58.1. Les propietats de JavaFX

Una propietat de JavaFX és essencialment una classe que encapsula un camp d'un d'objecte afegint la capacitat de notificar els canvis en els seus valors i permetent l'enllaç dinàmic entre més d'una propietat de manera que el canvi de valor d'una d'elles produex un canvi de valor en l'altra.

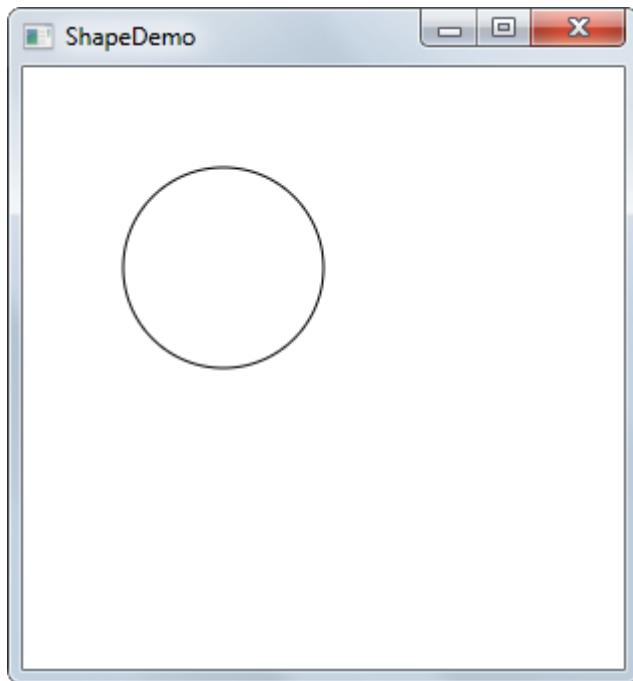
58.1.1. *Vincular propietats*

JavaFX permet vincular una propietat d'un objecte amb una propietat d'un segon objecte de manera que un canvi en la propietat origen produex un canvi en la propietat objectiu de forma transparent.

Diem que l'objecte origen és un objecte vinculable (bindable object) o observable i l'objecte objectiu és un objecte vinculant o observador (binding object)

Anem a veure un exemple d'utilització de la vinculació de propietats:

Si redimensionem la finestra de l'exemple anterior veurem que el cercle no queda centralitzat en compte les noves dimensions de la finestra.



Ens interessaria poder vincular les coordenades x i y del centre del cercle al punt central del Pane contenidor de manera que si el centre del *Pane* canvia les coordenades del centre del cercle canviem automàticament.

58.1.2. Implementació 1 - Utilitzant directament el patró observer

```
package josep.mavenproject1;

import javafx.application.Application;
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;

import javafx.stage.Stage;

public class App extends Application {
```

```

@Override
public void start(Stage primaryStage) {

    Pane pane = new Pane();

    Circle circle = new Circle();
    pane.widthProperty().addListener(new ChangeListener<Number>() {
        @Override
        public void changed(ObservableValue<? extends Number> ov,
        Number t, Number t1) {
            circle.setCenterX(t1.doubleValue() / 2);
        }
    });

    pane.heightProperty().addListener(new ChangeListener<Number>() {
        @Override
        public void changed(ObservableValue<? extends Number> ov,
        Number t, Number t1) {
            circle.setCenterY(t1.doubleValue() / 2);
        }
    });
    circle.setRadius(50);
    circle.setStroke(Color.BLACK);
    circle.setFill(Color.WHITE);
    pane.getChildren().add(circle);

    Scene scene = new Scene(pane, 300, 300);
    primaryStage.setTitle("Property Binding Demo");
    primaryStage.setScene(scene);
    primaryStage.show();
}

public static void main(String[] args) {
    launch(args);
}
}

```

58.1.3. Implementació 2 - Utilitzant bindings

```

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;

```

```
import javafx.stage.Stage;

public class PropertyBindingDemo extends Application {

    @Override
    public void start(Stage primaryStage) {

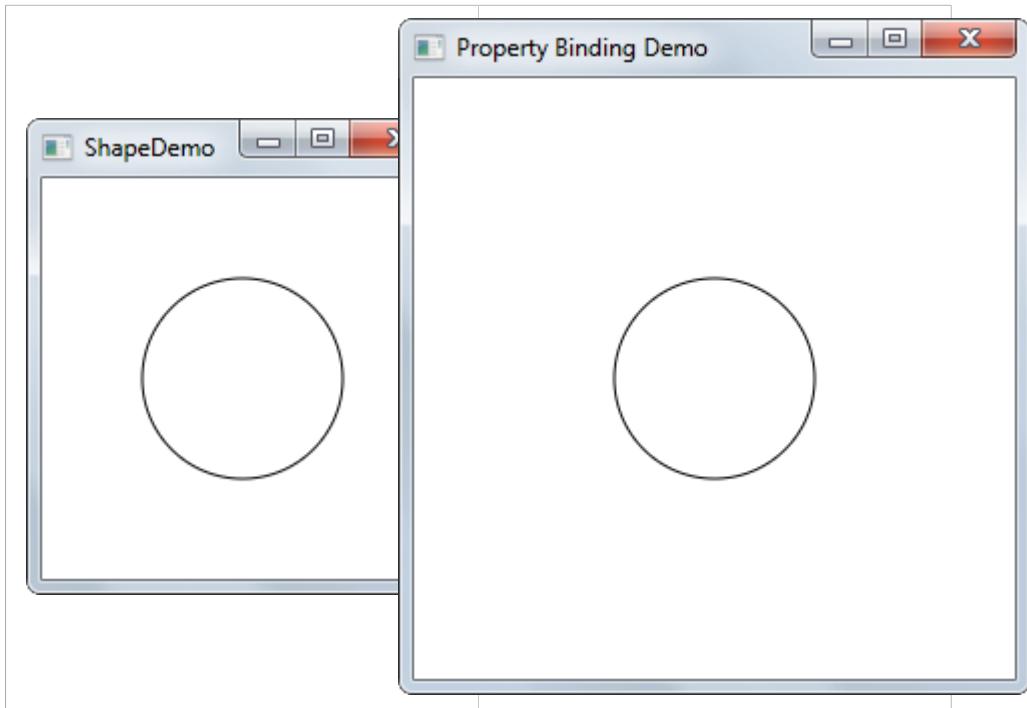
        Pane pane = new Pane();

        Circle circle = new Circle();
        circle.centerXProperty().bind(pane.widthProperty().divide(2)); ❶

        circle.centerYProperty().bind(pane.heightProperty().divide(2)); ❷
        circle.setRadius(50);
        circle.setStroke(Color.BLACK);
        circle.setFill(Color.WHITE);
        pane.getChildren().add(circle);

        Scene scene = new Scene(pane, 300, 300);
        primaryStage.setTitle("Property Binding Demo");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```



- ② Aquestes dues línies són les responsables de la vinculació.

58.1.4. Explicació de l'exemple

1. La classe **Circle** té les propietats **centerXProperty**, **centerYProperty** responsables de retornar la posició del centre del cercle. La majoria de les propietats en JavaFX són tant *observers* com *observable*.
2. Les propietats que son *observers* disposen del mètode **bind**. El mètode **bind** espera un paràmetre de tipus *observable*, en general una altra propietat.

El mètode **bind** vincula una propietat *observer* amb una propietat *observable* de la següent manera:

```
proprietat_observer.bind(proprietat_observable)
```

3. El mètode **bind** està definit a la interfície **javafx.beans.property.Property**, una propietat amb capacitats de "binding", un *observer*, implementa la interfície anterior.

4. Un objecte *observable* implementa la interfície **javafx.beans.value.ObservableValue**. Res presenten objectes que poden ser observats i per tant poden ser objectiu d'una vinculació.
5. JavaFX defineix propietats amb **capacitats de vinculació** per tots els **tipus primitius** i per les **cadenes**.

Les propietats amb **capacitats de vinculació** es reconeixen pel seu nom, totes acaben amb la paraula **Property**, per exemple, **DoubleProperty**, **LongProperty** o **StringProperty**.

6. Per **convenció**, les propietats amb capacitats de vinculació estan dotades d'un *getter* i d'un *setter*. Per exemple, si l'atribut que volem vincular s'anomenés **centerX** creariem els següents mètodes:

getCenterX()

Permet obtenir el valor de la propietat.

setCenterX(double value)

Permet assignar un valor a la propietat.

centerProperty()

Retorna la propietat en si, en aquest cas retornaria un **DoubleProperty**.

```
public class Circle {
    private DoubleProperty centerX;

    public double getCenterX() { ... }

    public void setCenterX(double value) { ... }

    public DoubleProperty centerXProperty() { ... }
}
```

7. Les propietats numèriques tenen mètodes per sumar, restar, multiplicar i dividir un valor en una propietat vinculada i retornar un nova propietat observable.

Per exemple, **pane.heightProperty().divide(2)** retorna una nova propietat observable que representa la meitat de l'amplada del *Pane*.



La instrucció
`circle.centerXProperty().bind(pane.widthProperty().divide(2));`
és equivalent a `centerX.bind(width.divide(2));`

Un altre exemple de vinculació de propietats.

```
import javafx.beans.property.DoubleProperty;
import javafx.beans.property.SimpleDoubleProperty;

public class BindingTest {

    public static void main(String[] args) {

        DoubleProperty d1 = new SimpleDoubleProperty(1); ①
        DoubleProperty d2 = new SimpleDoubleProperty(2);
        d1.bind(d2);
        System.out.println("d1 = " + d1.getValue() + ", d2 = " +
d2.getValue());
        d2.setValue(70.2);
        System.out.println("d1 = " + d1.getValue() + ", d2 = " +
d2.getValue());
    }
}
```

- ① La classe **DoubleProperty** és abstracte, la seva implementació esta a la subclasse **SimpleDoubleProperty**.



L'exemple anterior treballa amb una vinculació unidireccional.

Si les dues propietats vinculades són vinculables i observables és possible vincularles bidireccionalment amb l'ajuda del mètode **bindBidirectional()**.

58.1.5. Escoltar propietats

A vegades només voldrem estar pends d'un canvi d'una propietat i actuar en conseqüència enllloc de fer una vinculació completa i enllaçar dues propietats.

Les interfícies **Observable** i **ObservableValue** s'encarreguen de llançar les notificacions de canvi, les interfícies **InvalidationListener** i **ChangeListener**.

La diferència entre les dues és que la interfície **ObservableValue** encapsula un valor i envia els seus canvis als **ChangeListener** registrats previament, en canvi, la interfície **Observable** no encapsula cap valor i notifica als **OnvalidationListener** registrats.

Per exemple, el següent fragment de codi implementa un **TextField** que només accepta números.

```
TextField tf = new TextField();
tf.textProperty().addListener(new ChangeListener<String>() {
    @Override
    public void changed(ObservableValue<? extends String> observable,
    String oldValue, String newValue) {
        if (!newValue.matches("\\d*")) {
            tf.setText(newValue.replaceAll("[^\\d]", ""));
        }
    }
});
```

58.2. Interfície ObservableList

De forma similar a com s'escolten les propietats també es poden escoltar llistes de propietats mitjançant la interfície **ObservableList**.

Vegem-ho amb un exemple:

Model.

```
package listviewtest;

import java.util.Arrays;
import java.util.List;

public class ListViewTestModel {

    private List<String> ips;

    public ListViewTestModel() {
        ips = Arrays.asList(new String[]
{"912.168.2.1", "192.168.2.23", "192.168.2.103", "192.168.2.13", "192.168.2.109", "192.168.2.101", "192.168.2.102", "192.168.2.104", "192.168.2.105", "192.168.2.106", "192.168.2.107", "192.168.2.108", "192.168.2.100", "192.168.2.10", "192.168.2.20", "192.168.2.30", "192.168.2.40", "192.168.2.50", "192.168.2.60", "192.168.2.70", "192.168.2.80", "192.168.2.90"}, "192.168.2.11", "192.168.2.21", "192.168.2.31", "192.168.2.41", "192.168.2.51", "192.168.2.61", "192.168.2.71", "192.168.2.81", "192.168.2.91"}, "192.168.2.12", "192.168.2.22", "192.168.2.32", "192.168.2.42", "192.168.2.52", "192.168.2.62", "192.168.2.72", "192.168.2.82", "192.168.2.92"}, "192.168.2.13", "192.168.2.23", "192.168.2.33", "192.168.2.43", "192.168.2.53", "192.168.2.63", "192.168.2.73", "192.168.2.83", "192.168.2.93"}, "192.168.2.14", "192.168.2.24", "192.168.2.34", "192.168.2.44", "192.168.2.54", "192.168.2.64", "192.168.2.74", "192.168.2.84", "192.168.2.94"}, "192.168.2.15", "192.168.2.25", "192.168.2.35", "192.168.2.45", "192.168.2.55", "192.168.2.65", "192.168.2.75", "192.168.2.85", "192.168.2.95"}, "192.168.2.16", "192.168.2.26", "192.168.2.36", "192.168.2.46", "192.168.2.56", "192.168.2.66", "192.168.2.76", "192.168.2.86", "192.168.2.96"}, "192.168.2.17", "192.168.2.27", "192.168.2.37", "192.168.2.47", "192.168.2.57", "192.168.2.67", "192.168.2.77", "192.168.2.87", "192.168.2.97"}, "192.168.2.18", "192.168.2.28", "192.168.2.38", "192.168.2.48", "192.168.2.58", "192.168.2.68", "192.168.2.78", "192.168.2.88", "192.168.2.98"}, "192.168.2.19", "192.168.2.29", "192.168.2.39", "192.168.2.49", "192.168.2.59", "192.168.2.69", "192.168.2.79", "192.168.2.89", "192.168.2.99"}, "192.168.2.20", "192.168.2.30", "192.168.2.40", "192.168.2.50", "192.168.2.60", "192.168.2.70", "192.168.2.80", "192.168.2.90"}, "192.168.2.21", "192.168.2.31", "192.168.2.41", "192.168.2.51", "192.168.2.61", "192.168.2.71", "192.168.2.81", "192.168.2.91"}, "192.168.2.22", "192.168.2.32", "192.168.2.42", "192.168.2.52", "192.168.2.62", "192.168.2.72", "192.168.2.82", "192.168.2.92"}, "192.168.2.23", "192.168.2.33", "192.168.2.43", "192.168.2.53", "192.168.2.63", "192.168.2.73", "192.168.2.83", "192.168.2.93"}, "192.168.2.24", "192.168.2.34", "192.168.2.44", "192.168.2.54", "192.168.2.64", "192.168.2.74", "192.168.2.84", "192.168.2.94"}, "192.168.2.25", "192.168.2.35", "192.168.2.45", "192.168.2.55", "192.168.2.65", "192.168.2.75", "192.168.2.85", "192.168.2.95"}, "192.168.2.26", "192.168.2.36", "192.168.2.46", "192.168.2.56", "192.168.2.66", "192.168.2.76", "192.168.2.86", "192.168.2.96"}, "192.168.2.27", "192.168.2.37", "192.168.2.47", "192.168.2.57", "192.168.2.67", "192.168.2.77", "192.168.2.87", "192.168.2.97"}, "192.168.2.28", "192.168.2.38", "192.168.2.48", "192.168.2.58", "192.168.2.68", "192.168.2.78", "192.168.2.88", "192.168.2.98"}, "192.168.2.29", "192.168.2.39", "192.168.2.49", "192.168.2.59", "192.168.2.69", "192.168.2.79", "192.168.2.89", "192.168.2.99"}, "192.168.2.30", "192.168.2.40", "192.168.2.50", "192.168.2.60", "192.168.2.70", "192.168.2.80", "192.168.2.90"}, "192.168.2.31", "192.168.2.41", "192.168.2.51", "192.168.2.61", "192.168.2.71", "192.168.2.81", "192.168.2.91"}, "192.168.2.32", "192.168.2.42", "192.168.2.52", "192.168.2.62", "192.168.2.72", "192.168.2.82", "192.168.2.92"}, "192.168.2.33", "192.168.2.43", "192.168.2.53", "192.168.2.63", "192.168.2.73", "192.168.2.83", "192.168.2.93"}, "192.168.2.34", "192.168.2.44", "192.168.2.54", "192.168.2.64", "192.168.2.74", "192.168.2.84", "192.168.2.94"}, "192.168.2.35", "192.168.2.45", "192.168.2.55", "192.168.2.65", "192.168.2.75", "192.168.2.85", "192.168.2.95"}, "192.168.2.36", "192.168.2.46", "192.168.2.56", "192.168.2.66", "192.168.2.76", "192.168.2.86", "192.168.2.96"}, "192.168.2.37", "192.168.2.47", "192.168.2.57", "192.168.2.67", "192.168.2.77", "192.168.2.87", "192.168.2.97"}, "192.168.2.38", "192.168.2.48", "192.168.2.58", "192.168.2.68", "192.168.2.78", "192.168.2.88", "192.168.2.98"}, "192.168.2.39", "192.168.2.49", "192.168.2.59", "192.168.2.69", "192.168.2.79", "192.168.2.89", "192.168.2.99"}, "192.168.2.40", "192.168.2.50", "192.168.2.60", "192.168.2.70", "192.168.2.80", "192.168.2.90"}, "192.168.2.41", "192.168.2.51", "192.168.2.61", "192.168.2.71", "192.168.2.81", "192.168.2.91"}, "192.168.2.42", "192.168.2.52", "192.168.2.62", "192.168.2.72", "192.168.2.82", "192.168.2.92"}, "192.168.2.43", "192.168.2.53", "192.168.2.63", "192.168.2.73", "192.168.2.83", "192.168.2.93"}, "192.168.2.44", "192.168.2.54", "192.168.2.64", "192.168.2.74", "192.168.2.84", "192.168.2.94"}, "192.168.2.45", "192.168.2.55", "192.168.2.65", "192.168.2.75", "192.168.2.85", "192.168.2.95"}, "192.168.2.46", "192.168.2.56", "192.168.2.66", "192.168.2.76", "192.168.2.86", "192.168.2.96"}, "192.168.2.47", "192.168.2.57", "192.168.2.67", "192.168.2.77", "192.168.2.87", "192.168.2.97"}, "192.168.2.48", "192.168.2.58", "192.168.2.68", "192.168.2.78", "192.168.2.88", "192.168.2.98"}, "192.168.2.49", "192.168.2.59", "192.168.2.69", "192.168.2.79", "192.168.2.89", "192.168.2.99"}, "192.168.2.50", "192.168.2.60", "192.168.2.70", "192.168.2.80", "192.168.2.90"}, "192.168.2.51", "192.168.2.61", "192.168.2.71", "192.168.2.81", "192.168.2.91"}, "192.168.2.52", "192.168.2.62", "192.168.2.72", "192.168.2.82", "192.168.2.92"}, "192.168.2.53", "192.168.2.63", "192.168.2.73", "192.168.2.83", "192.168.2.93"}, "192.168.2.54", "192.168.2.64", "192.168.2.74", "192.168.2.84", "192.168.2.94"}, "192.168.2.55", "192.168.2.65", "192.168.2.75", "192.168.2.85", "192.168.2.95"}, "192.168.2.56", "192.168.2.66", "192.168.2.76", "192.168.2.86", "192.168.2.96"}, "192.168.2.57", "192.168.2.67", "192.168.2.77", "192.168.2.87", "192.168.2.97"}, "192.168.2.58", "192.168.2.68", "192.168.2.78", "192.168.2.88", "192.168.2.98"}, "192.168.2.59", "192.168.2.69", "192.168.2.79", "192.168.2.89", "192.168.2.99"}, "192.168.2.60", "192.168.2.70", "192.168.2.80", "192.168.2.90"}, "192.168.2.61", "192.168.2.71", "192.168.2.81", "192.168.2.91"}, "192.168.2.62", "192.168.2.72", "192.168.2.82", "192.168.2.92"}, "192.168.2.63", "192.168.2.73", "192.168.2.83", "192.168.2.93"}, "192.168.2.64", "192.168.2.74", "192.168.2.84", "192.168.2.94"}, "192.168.2.65", "192.168.2.75", "192.168.2.85", "192.168.2.95"}, "192.168.2.66", "192.168.2.76", "192.168.2.86", "192.168.2.96"}, "192.168.2.67", "192.168.2.77", "192.168.2.87", "192.168.2.97"}, "192.168.2.68", "192.168.2.78", "192.168.2.88", "192.168.2.98"}, "192.168.2.69", "192.168.2.79", "192.168.2.89", "192.168.2.99"}, "192.168.2.70", "192.168.2.80", "192.168.2.90"}, "192.168.2.71", "192.168.2.81", "192.168.2.91"}, "192.168.2.72", "192.168.2.82", "192.168.2.92"}, "192.168.2.73", "192.168.2.83", "192.168.2.93"}, "192.168.2.74", "192.168.2.84", "192.168.2.94"}, "192.168.2.75", "192.168.2.85", "192.168.2.95"}, "192.168.2.76", "192.168.2.86", "192.168.2.96"}, "192.168.2.77", "192.168.2.87", "192.168.2.97"}, "192.168.2.78", "192.168.2.88", "192.168.2.98"}, "192.168.2.79", "192.168.2.89", "192.168.2.99"}, "192.168.2.80", "192.168.2.90"}, "192.168.2.81", "192.168.2.91"}, "192.168.2.82", "192.168.2.92"}, "192.168.2.83", "192.168.2.93"}, "192.168.2.84", "192.168.2.94"}, "192.168.2.85", "192.168.2.95"}, "192.168.2.86", "192.168.2.96"}, "192.168.2.87", "192.168.2.97"}, "192.168.2.88", "192.168.2.98"}, "192.168.2.89", "192.168.2.99"}, "192.168.2.90"}, "192.168.2.91"}, "192.168.2.92"}, "192.168.2.93"}, "192.168.2.94"}, "192.168.2.95"}, "192.168.2.96"}, "192.168.2.97"}, "192.168.2.98"}, "192.168.2.99"});
```

```
public List<String> getLlistaIP() {  
    return ips;  
}  
  
public String toString() {  
  
    return Arrays.toString(ips.toArray());  
}  
}
```

Vista.

```
package listviewtest;  
  
import javafx.scene.control.ListView;  
import javafx.scene.control.TextArea;  
import javafx.scene.layout.VBox;  
  
public class ListViewTestView {  
  
    private VBox rootPane;  
    private ListView<String> listView;  
    private TextArea textArea;  
  
    public ListViewTestView() {  
        listView = new ListView<>();  
        textArea = new TextArea();  
        rootPane = new VBox(listView, textArea);  
    }  
  
    protected VBox getRootPane() {  
        return rootPane;  
    }  
  
    protected ListView<String> getListView() {  
        return listView;  
    }  
  
    protected TextArea getTextArea() {  
        return textArea;  
    }  
}
```

Controlador.

```

package listviewtest;

import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;

public class ListViewTestController {

    private ListViewTestModel model;
    private ListViewTestView view;

    public ListViewTestController(ListViewTestModel m, ListViewTestView v) {
        model = m;
        view = v;
        initView();
    }

    public void initView() {
        ObservableList<String> ips =
            FXCollections.observableArrayList(model.getListaIP()); ①
        view.getListView().setItems(ips);
    }

    public void initController() {

        view.getListView().getSelectionModel().selectedItemProperty().addListener(new
            ChangeListener<String>() { ②
                @Override
                public void changed(ObservableValue<? extends String>
                    observable, String oldValue, String newValue) {
                    view.getTextArea().setText(newValue);
                }
            });
    }
}

```

① Encapsuem la **List** dins d'una **ObservableList**.

② Accedim als esdeveniments dels items de la llista.

Main.

```
package listviewtest;
```

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class ListViewTest extends Application {

    @Override
    public void start(Stage primaryStage
    ) {
        ListViewTestModel m = new ListViewTestModel();
        ListViewTestView v = new ListViewTestView();
        ListViewTestController c = new ListViewTestController(m, v);
        c.initController();

        primaryStage.setTitle("ListView Test");
        primaryStage.setScene(new Scene(v.getRootPane(), 300, 400));
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

58.3. ObservableList i TableView

TODO

58.4. Exemple - TableView

module-info.java.

```
module tableviewdemo {  
    requires javafx.controls;  
    exports tableviewdemo;  
    opens tableviewdemo.models;  
}
```

tableviewdemo.App.java.

```
package tableviewdemo;  
  
import javafx.application.Application;  
import static javafx.application.Application.launch;  
import javafx.collections.ObservableList;  
import javafx.scene.Scene;  
import javafx.stage.Stage;  
import tableviewdemo.controllers.TableViewDemoController;  
import tableviewdemo.models.Contacate;  
import tableviewdemo.models.dao.OrigenDades;  
import tableviewdemo.views.TableViewDemoView;  
  
public class App extends Application {  
  
    @Override  
    public void start(Stage primaryStage) {  
        OrigenDades dao = new OrigenDades();  
        ObservableList<Contacate> contactes = dao.getContactes();  
        TableViewDemoView v = new TableViewDemoView();  
        TableViewDemoController c = new  
        TableViewDemoController(contactes, v);  
        c.initController();  
  
        primaryStage.setTitle("TableView Demo");  
        primaryStage.setScene(new Scene(v.getRootPane(), 500, 500));  
        primaryStage.show();  
    }  
  
    public static void main(String[] args) {  
        launch(args);  
    }  
}
```

tableviewdemo.views.TableViewDemoView.java.

```
package tableviewdemo.views;

import javafx.geometry.Insets;
import javafx.scene.control.Label;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableView;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.text.Font;

public class TableViewDemoView {

    private VBox rootPane;
    private Label contactesLabel;
    private TableView contactesTableView;
    private TableColumn nomTableColumn;
    private TableColumn cognomsTableColumn;
    private TableColumn correuTableColumn;
    private TableColumn actiuTableColumn;
    private HBox nouRegistreVBox;

    public TableViewDemoView() {
        contactesLabel = new Label("Contactes");
        contactesLabel.setFont(new Font("Arial", 20));

        contactesTableView = new TableView();
        contactesTableView.setEditable(true);

        nomTableColumn = new TableColumn("Nom");
        nomTableColumn.setMinWidth(100);
        cognomsTableColumn = new TableColumn("Cognoms");
        cognomsTableColumn.setMinWidth(200);
        correuTableColumn = new TableColumn("Correu");
        correuTableColumn.setMaxWidth(200);
        actiuTableColumn = new TableColumn("Actiu");
        actiuTableColumn.setMaxWidth(30);

        contactesTableView.getColumns().addAll(nomTableColumn,
        cognomsTableColumn, correuTableColumn, actiuTableColumn);

        rootPane = new VBox();
        rootPane.setSpacing(5);
        rootPane.setPadding(new Insets(10, 10, 10, 10));
    }
}
```

```
    rootPane.getChildren().addAll(contactesLabel,
contactesTableView);

}

public VBox getRootPane() {
    return rootPane;
}

public Label getContactesLabel() {
    return contactesLabel;
}

public TableView getContactesTableView() {
    return contactesTableView;
}

public TableColumn getNomTableColumn() {
    return nomTableColumn;
}

public TableColumn getCognomsTableColumn() {
    return cognomsTableColumn;
}

public TableColumn getCorreuTableColumn() {
    return correuTableColumn;
}

public HBox getNouRegistreVBox() {
    return nouRegistreVBox;
}

public TableColumn getActiuTableColumn() {
    return actiuTableColumn;
}
}
```

tableviewdemo.models.Contacte.

```
package tableviewdemo.models;

import javafx.beans.property.SimpleBooleanProperty;
import javafx.beans.property.SimpleStringProperty;
```

```
public class Contacte {

    private final SimpleStringProperty nom;
    private final SimpleStringProperty cognoms;
    private final SimpleStringProperty correu;
    private final SimpleBooleanProperty actiu;

    public Contacte(String nom, String cognoms, String email, boolean actiu) {
        this.nom = new SimpleStringProperty(nom);
        this.cognoms = new SimpleStringProperty(cognoms);
        this.correu = new SimpleStringProperty(email);
        this.actiu = new SimpleBooleanProperty(actiu);
    }

    public String getNom() {
        return nom.get();
    }

    public SimpleStringProperty nomProperty() {
        return nom;
    }

    public void setNom(String nom) {
        this.nom.setValue(nom);
    }

    public String getCognoms() {
        return cognoms.get();
    }

    public SimpleStringProperty cognomsProperty() {
        return cognoms;
    }

    public void setCognoms(String cognoms) {
        this.cognoms.setValue(cognoms);
    }

    public String getEmail() {
        return correu.get();
    }

    public SimpleStringProperty emailProperty() {
        return correu;
```

```
}

public void setEmail(String email) {
    this.correu.setValue(email);
}

public boolean isActiu() {
    return actiu.get();
}

public SimpleBooleanProperty actiuProperty() {
    return actiu;
}

public void setActiu(boolean actiu) {
    this.actiu.setValue(actiu);
}

}
```

tableviewdemo.models.dao.OrigenDades.

```
package tableviewdemo.models.dao;

import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import tableviewdemo.models.Contacte;

public class OrigenDades {

    public ObservableList<Contacte> getContactes() {
        ObservableList<Contacte> ret =
FXCollections.observableArrayList();

        ret.add(new Contacte("nom 1", "cognoms 1", "correu1@test.com",
true));
        ret.add(new Contacte("nom 2", "cognoms 2", "correu2@test.com",
true));
        ret.add(new Contacte("nom 3", "cognoms 3", "correu3@test.com",
false));
        ret.add(new Contacte("nom 4", "cognoms 4", "correu4@test.com",
true));
        ret.add(new Contacte("nom 5", "cognoms 5", "correu5@test.com",
false));
    }
}
```

```

        return ret;
    }
}

```

tableviewdemo.controllers.TableViewDemoController.

```

package tableviewdemo.controllers;

import javafx.collections.ObservableList;

import javafx.scene.control.cell.CheckBoxTableCell;
import javafx.scene.control.cell.PropertyValueFactory;
import tableviewdemo.models.Contacete;
import tableviewdemo.views.TableViewDemoView;

public class TableViewDemoController {

    private ObservableList<Contacete> model;
    private TableViewDemoView view;

    public TableViewDemoController(ObservableList<Contacete> m,
TableViewDemoView v) {
        model = m;
        view = v;
        initView();
    }

    public void initView() {
        view.getNomTableColumn().setCellValueFactory(new
PropertyValueFactory<Contacete, String>("nom")); // Extreu el nom de
getNom
        view.getCognomsTableColumn().setCellValueFactory(new
PropertyValueFactory<Contacete, String>("cognoms"));
        view.getCorreuTableColumn().setCellValueFactory(new
PropertyValueFactory<Contacete, String>("email"));
        view.getActiuTableColumn().setCellValueFactory(new
PropertyValueFactory<Contacete, Boolean>("actiu"));

        view.getActiuTableColumn().setCellFactory(CheckBoxTableCell.forTableColumn(view.get
//}});
        // O passem la llista
        view.getContactesTableView().setItems(model);

        // O passem cadascún dels elements
    }
}

```

Exemple - TableView

```
//      for(Contacte c : model){  
//          view.getContactesTableView().getItems().add(c);  
//      }  
  
public void initController() {  
    // TODO: Implementar totes les classes EventHandler  
}  
  
}
```

59

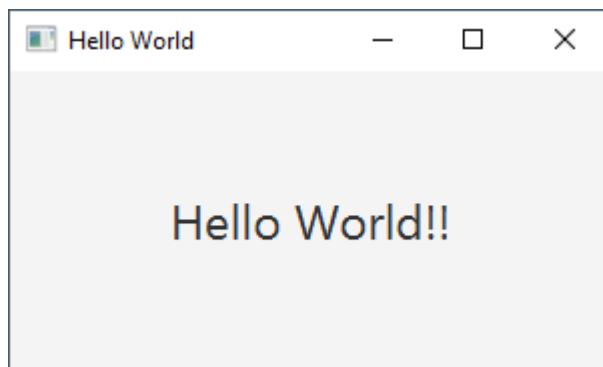
JavaFX - FXML

JavaFX FXML és un format que permet elaborar interfícies gràfiques d'usuari en XML separant automàticament el codi de la vista de la resta.

59.1. Creació d'una vista

Per crear una vista en FXML cal crear un document de text d'extensió **.fxml** i començar a declarar la vista.

El següent exemple defineix un **VBox** amb una **Label** com a node fill:



hello.fxml.

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Label?> ①
<?import javafx.scene.layout.VBox?>
<?import javafx.scene.text.Font?>
```

```
<VBox alignment="CENTER" spacing="10.0" xmlns="http://javafx.com/
javafx/8.0.171" xmlns:fx="http://javafx.com/fxml/1"> ②
    <children> ③
        <Label alignment="CENTER"
prefHeight="150.0" prefWidth="300.0" text="Hello World!!">
            <font>
                <Font size="24.0" />
            </font>
        </Label>
    </children>
</VBox>
```

- ① Importem les classes utilitzades.
- ② Declarem el node arrel.
- ③ Afegim elements a la col·lecció **children** del node arrel.



Si fem referència a una que els "accessor" des de fxml, en Java començaríem per **get** o per **set**, però en fxml **no** es posen aquests prefixes.

59.2. Carregar un fitxer FXML

Un cop definida la vista en **fxml** cal un mecanisme per carregar-la i convertir-la en un objecte que es pugui manipular.

El següent codi mostra com fer-ho:

```
package hellofx;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class HelloFX extends Application {

    @Override
    public void start(Stage stage) throws Exception {
        FXMLLoader loader = new FXMLLoader(); ①
```

```

        Parent root =
loader.load(getClass().getResource("HelloFX.fxml")); ②

        Scene scene = new Scene(root);

        stage.setTitle("Hello World");
        stage.setScene(scene);
        stage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }

}

```

- ① La classe **FXMLLoader** permet carregar fitxers amb extensió **FXML**.
- ② A través d'un **FXMLLoader** carreguem la vista amb el mètode **load**. El mètode **getResource** retorna una URL a partir del nom del fitxer.

59.3. Afegir un controlador

Si volem manipular els nodes de la vista des de codi o bé volem gestionar els esdeveniments que s'hi generen, ens farà una classe que faci el paper de controlador i una manera de fer referència als nodes de la vista.

59.3.1. Crear la classe controlador

Podem crear una classe controlador com al següent exemple:

HelloFXController.java.

```

package hellofx;

import java.net.URL;
import java.util.ResourceBundle;
import javafx.fxml.Initializable;

public class HelloFXController implements Initializable { ①

    @Override
    public void initialize(URL url, ResourceBundle rb) { ②

```

```
// TODO
}
}
```

- ➊ Els controladors implementen la interfície **Initializable**.
- ➋ La interfície **Initializable** obliga a implementar el mètode **initialize**. Aquest mètode es crida quan la vista ha estat creada completament i tenim accés a tots els seus nodes.



Accedir a un node de la vista al constructor del controlador donaria problemes perquè és possible que aquest node encara no existeixi.

59.3.2. Posar nom als nodes de la vista

Com que volem accedir als nodes de la vista des del controlador caldrà identificar-los, per fer-ho s'utilitza l'atribut **id** per a cadascun dels nodes als que es vulgui accedir des del controlador, per exemple:

hello.fxml.

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Label?>
<?import javafx.scene.layout.VBox?>
<?import javafx.scene.text.Font?>

<VBox alignment="CENTER" spacing="10.0" xmlns="http://
javafx.com/javafx/8.0.171" xmlns:fx="http://javafx.com/
fxml/1" fx:controller="hellofx.HelloFXController">
    <children>
        <Label fx:id="helloLabel" alignment="CENTER"
prefHeight="150.0" prefWidth="300.0" text="Hello World!!"> ➊
            <font>
                <Font size="24.0" />
            </font>
        </Label>
    </children>
</VBox>
```

- ➊ Establim que el nom de la **Label** és **helloLabel**.

59.3.3. Crear variables al controlador pels nodes de la vista

Un cop em assignat noms als nodes de la vista volem poder accedir-hi des de la classe controlador. Per fer-ho caldrà una variable del tipus de node adequat.

La manera de d'assignar variables al controlador que facin referència a nodes de la vista és:

1. Crear una variable a nivell de classe per a cada node de la vista.
2. Utilitzar el mateix nom de variable que el nom assignat a l'atribut **id** de la vista.
3. El tipus de dades de la variable ha de ser compatible amb l'element declarat a la vista.
4. Cal decorar la variable amb l'anotació **@FXML** per indicar a Java que la creació de la variable vindrà donada per la vista associada.

HelloFXController.java.

```
package hellofx;

import java.net.URL;
import java.util.ResourceBundle;
import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.control.Label;

public class HelloFXController implements Initializable {

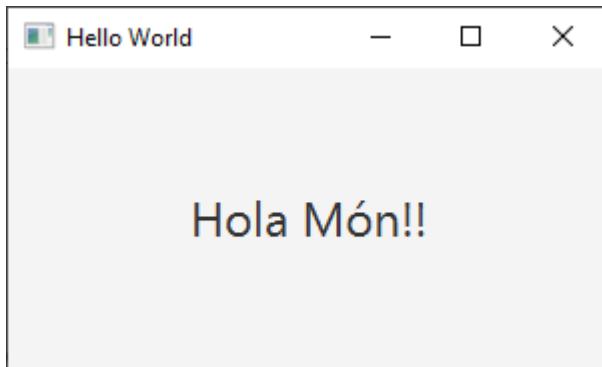
    @FXML ①
    private Label helloLabel; ②

    @Override
    public void initialize(URL url, ResourceBundle rb) {
        helloLabel.setText("Hola Món!!"); ③
    }
}
```

- ① Decorem la declaració de la variable amb l'anotació **@FXML**.
- ② El tipus de la variable (**Label**) coincideix amb l'element de la vista (**Label**) i el nom de la variable coincideix amb el **id** assignat al node a la vista.

- ③ Podem accedir als nodes de la vista a través de les variables declarades al controlador.

L'aplicació quedaría:



59.3.4. Gestió d'esdeveniments

Si volem gestionar els esdeveniments generats a través dels nodes de la vista caldrà:

1. Tenir accés als nodes implicats des del controlador.
2. Proporcionar un mètode amb el codi que s'executarà en resposta a l'esdeveniment, la signatura d'aquest mètode és, **metode(Event: e)** on **e** és un objecte derivat de la classe **Event**.
3. El mètode anterior ha d'estar decorat amb l'anotació **@FXML**.
4. A la vista caldrà indicar en algun dels atributs *onEsdeveniment* quin és el mètode que es vincula a l'esdeveniment, el nom del mètode va precedit de **#**.

Vegem-ho amb un exemple:

A la vista

hello.fxml.

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.layout.VBox?>
<?import javafx.scene.text.Font?>
```

```

<VBox alignment="CENTER" spacing="10.0" xmlns="http://
javafx.com/javafx/8.0.171" xmlns:fx="http://javafx.com/
fxml/1" fx:controller="hellofx.HelloFXController">
    <children>

        <Label fx:id="helloLabel" alignment="CENTER" prefHeight="150.0" prefWidth="300
World!!">
            <font>
                <Font size="24.0" />
            </font>
        </Label>
        <Button fx:id="helloButton" onAction="#sayHello" text="I en
❶ català?" />
    </children>
</VBox>

```

- ❶ Responem al esdeveniment **onAction** amb la crida del mètode **sayHello**.

Al controlador

HelloFXController.java.

```

package hellofx;

import java.net.URL;
import java.util.ResourceBundle;
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.control.Button;
import javafx.scene.control.Label;

public class HelloFXController implements Initializable {

    @FXML
    private Label helloLabel;
    @FXML
    private Button helloButton; ❶

    @Override
    public void initialize(URL url, ResourceBundle rb) {
        // TODO
    }
}

```

```
@FXML  
private void sayHello(ActionEvent event) { ②  
    helloLabel.setText("Hola Món!!");  
}  
}
```

- ① El node que genera l'esdeveniment ha de ser accessible.
- ② Creem un mètode capaç de gestionar l'esdeveniment, el nom del mètode coincideix amb el declarat a la vista i el tipus d'esdeveniment es compatible amb l'atribut **onEsdevenoment** utilitzat a la vista.

Aquest mètode ha d'estar decorat amb l'anotació **@FXML**.



60

JavaFX FXML - Un exemple guiat

En aquest apartat construirem una aplicació completa utilitzant vistes **fxml**.

Per construir les vistes utilitzarem un software anomenat **SceneBuilder** que permet generar els fitxers **fxml** arrossegant els controls des d'una interfície gràfica.

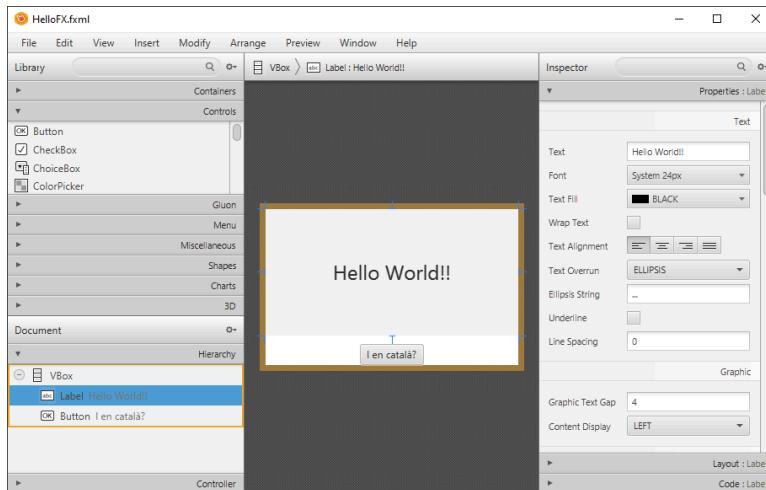


Figura 60.1. SceneBuilder en acció

Podeu baixar la versió del SceneBuilder compatible amb la versió de Java que utilitzeu al següent enllaç <https://gluonhq.com/products/scene-builder/>.

60.1. Implementació de la Vista

60.1.1. Paquets

Per fer l'aplicació **Contactes** utilitzarem una arquitectura Model-Vista-Controlador (MVC). Aquesta arquitectura promou la divisió del nostre codi en tres apartats clarament definits, un per cada element de l'arquitectura. A Java aquesta separació s'aconsegueix mitjançant la creació de tres paquets separats.

Treballarem amb els tres paquets següents:

contactes

contindrà la majoria de classes de control

contactes.models

contindrà les classes del model

contactes.vistes

contindrà les vistes

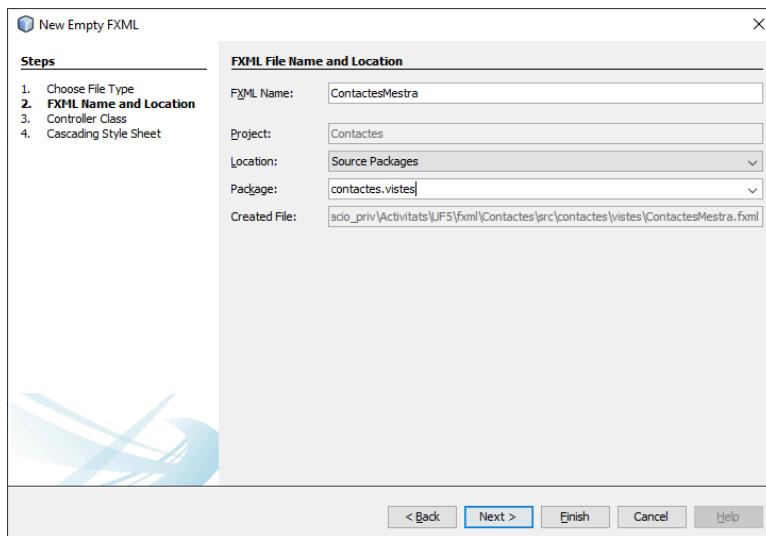
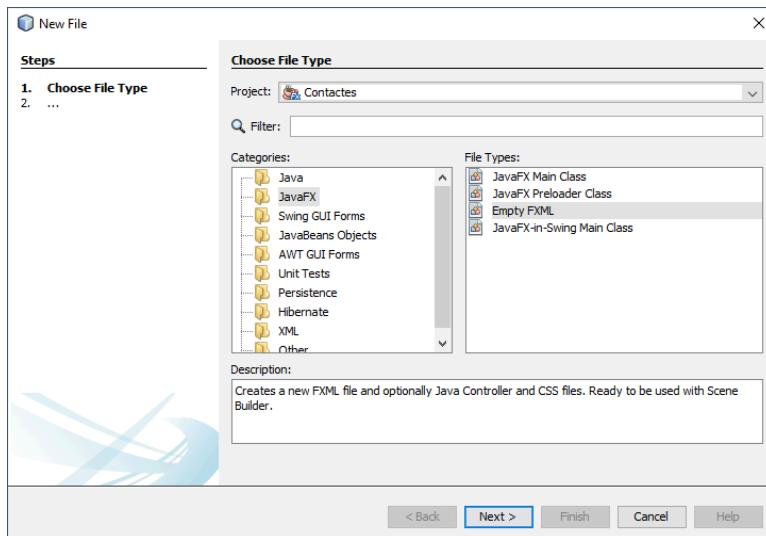


El paquet dedicat a les vistes contindrà també alguns controladors dedicats exclusivament a una vista. Els anomenarem **controladors-vista**.

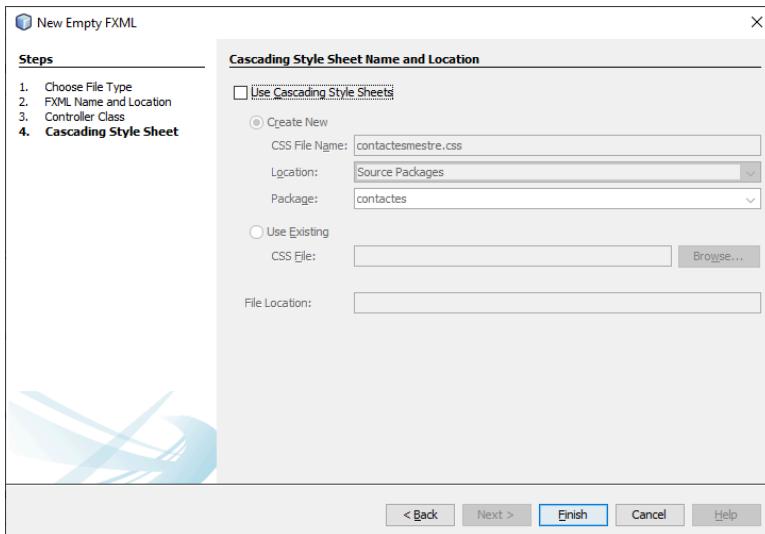
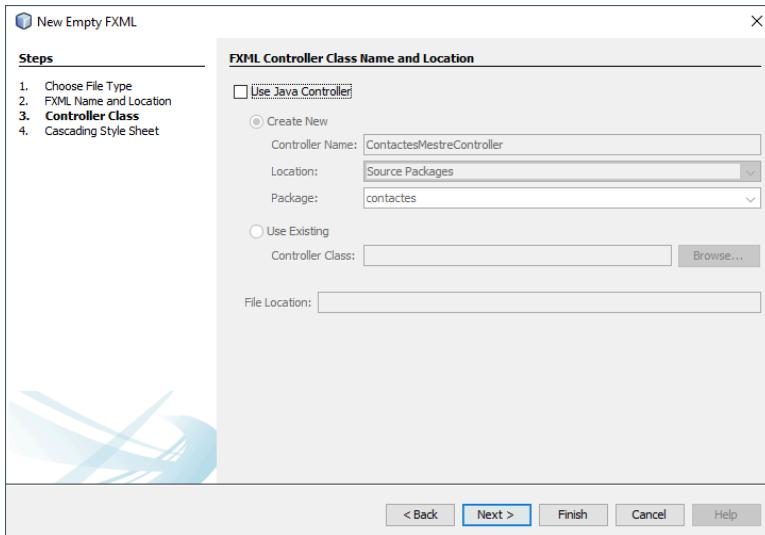
60.1.2. Formulari ContactesMestre

Anem a crear el formulari **ContactesMestre**, per fer-ho afegirem al projecte un fitxer **FXML**.

Formulari ContactesMestre

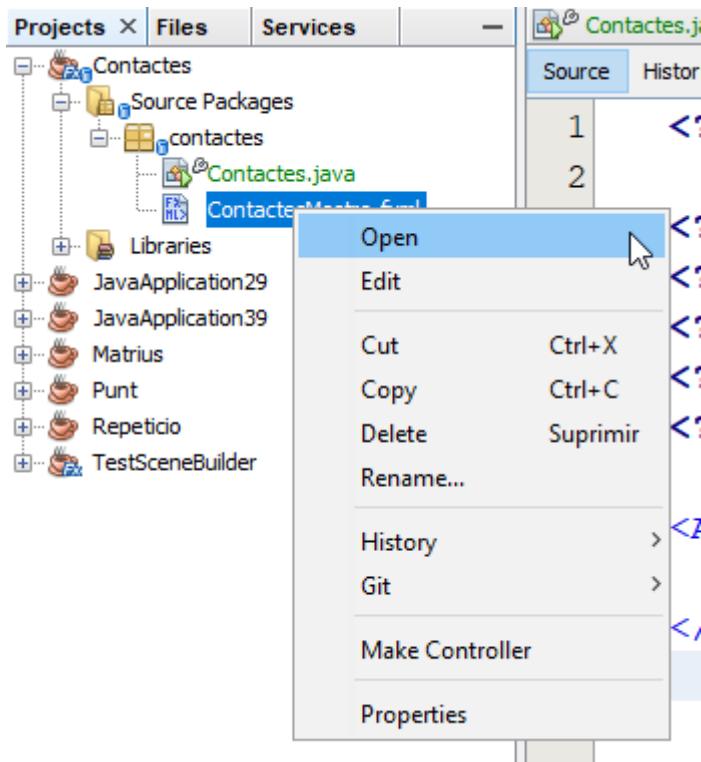


I de moment no afegirem ni **controlador** ni **full d'estil**.



60.1.3. Scene Builder

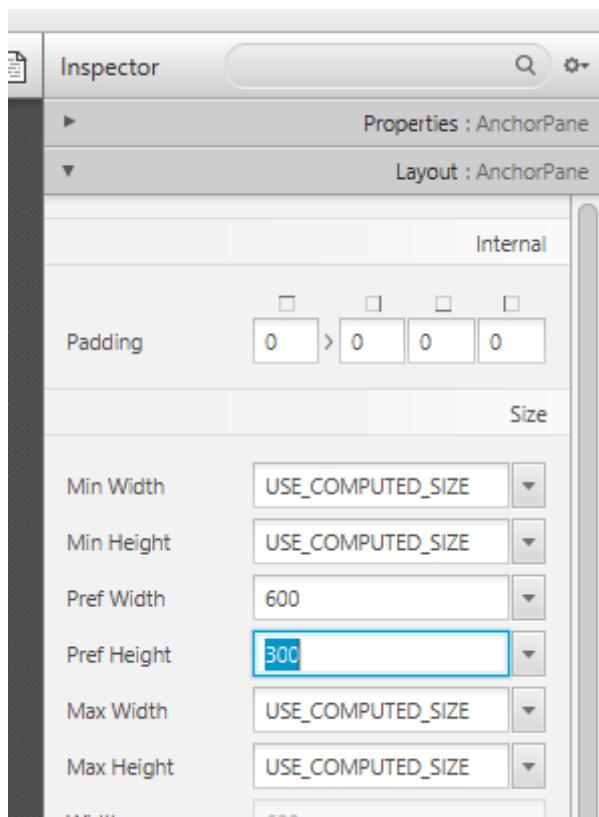
Un cop creat el fitxer **ContactesMestre.fxml** l'obrim amb el **Scene Builder**, botó dret sobre el fitxer a la finestra de projecte → Open.



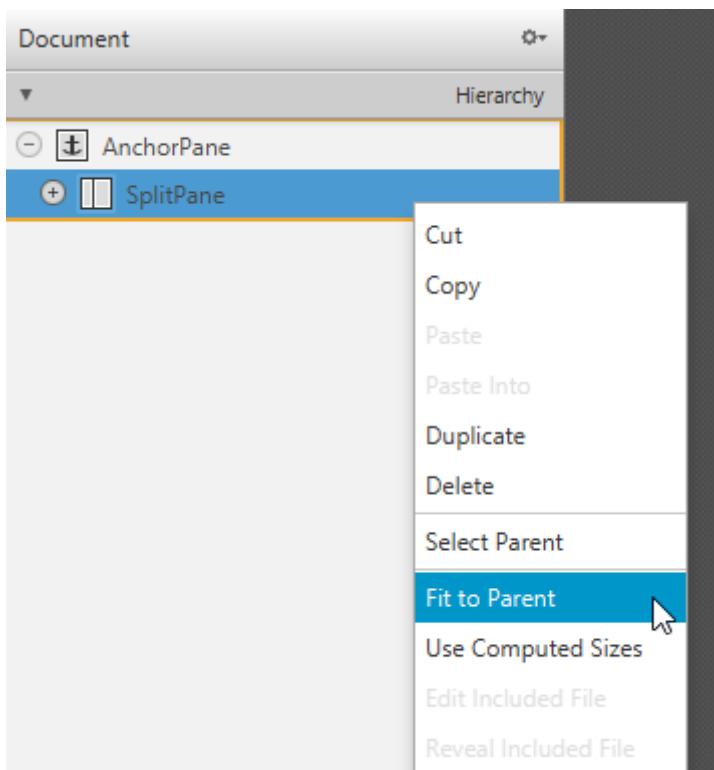
60.1.4. Disseny del formulari ContactesMestre

Hauries de veure el Scene Builder amb un AnchorPane (visible en la jerarquia de components (eina Hierarchy) situada a l'esquerra).

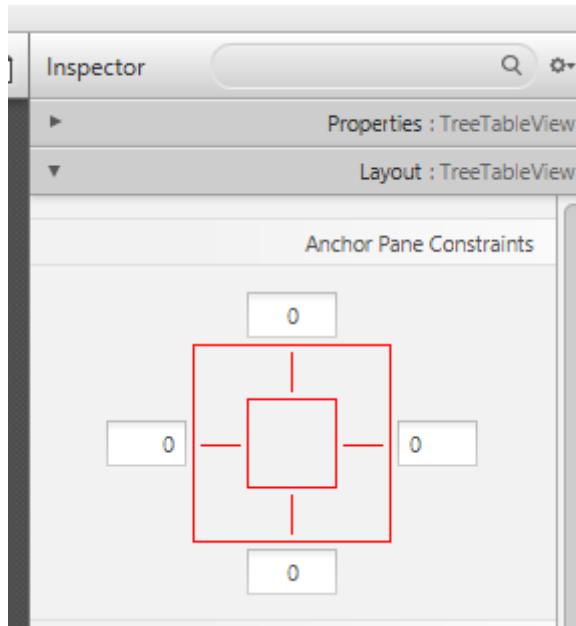
1. Selecciona el **AnchorPane** en la teva jerarquia i ajusta la grandària en l'apartat Layout (a la dreta):



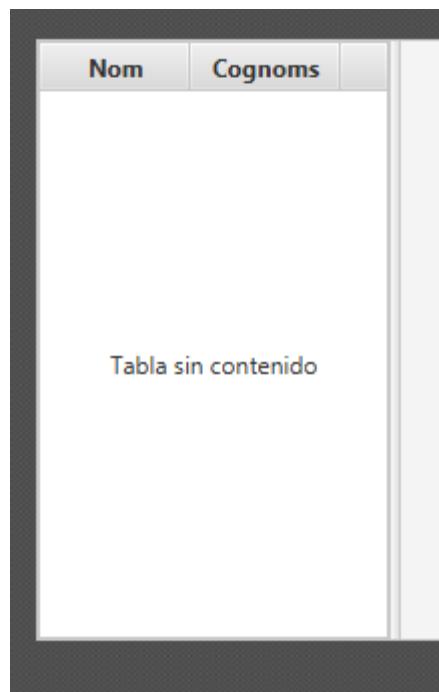
2. Afegeix un SplitPane (Horizontal) arrosegant-ho des de la biblioteca (Library) a l'àrea principal d'edició. Fes clic dret sobre el SplitPane en la jerarquia i tria Fit to Parent.



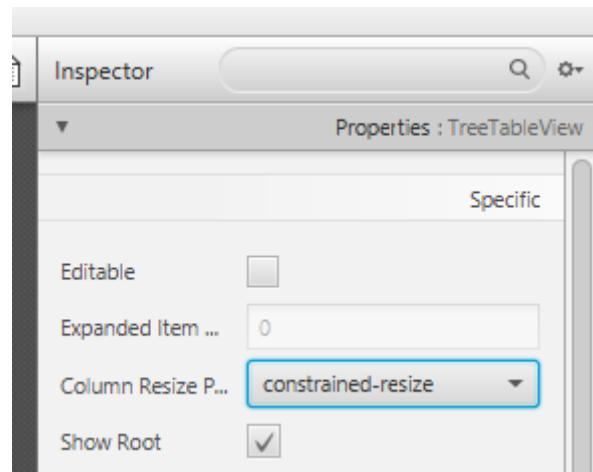
3. Arrossegaa un **TableView** (pestanya Controls) al costat esquerre del **SplitPane**. Selecciona la **TableView** (no una sola columna) i estableix les següents restriccions d'aparença (Layout) per a la **TableView**. Dins d'un AnchorPane sempre es poden establir ancoratges (anchors) per a les quatre voreres.



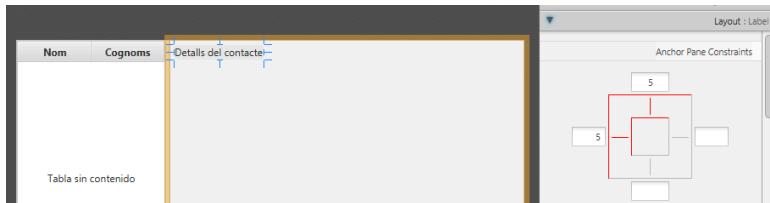
4. Ves al menú **Preview | Show Preview in Window** per a comprovar si es visualitza correctament. Intenta canviar la grandària de la finestra. La **TableView** hauria d'ajustar la seva grandària a la grandària de la finestra, perquè està "ancorada" a les seves vores.
5. Canvia el text de les columnes (pestanya Properties) a "Nom" i "Cognoms".



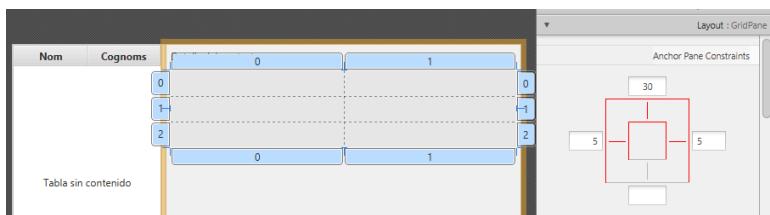
6. Selecciona la **TableView** i tria **constrained-resize** per la propietat **Column Resize Policy**. Això assegura que les columnes usaran sempre tot l'espai que tinguin disponible.



7. Afegeix una Label al costat dret del SplitPane amb el text "Detalls del contacte".



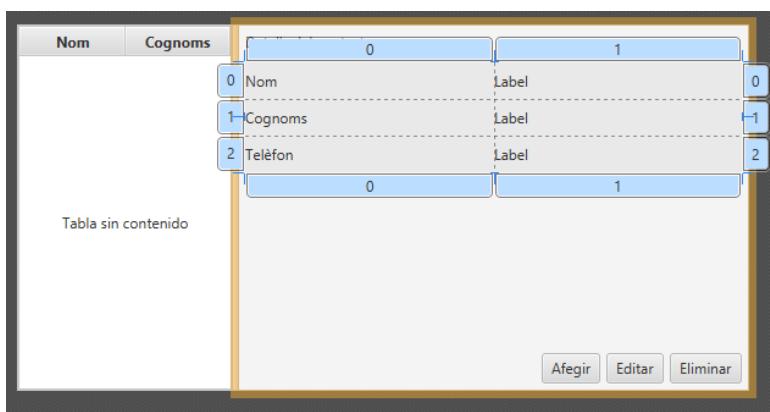
8. Afegeix un **GridPane** al costat dret, selecciona'l i ajusta la seva aparença usant ancoratges (superior, dret i esquerre).



9. Afegeix les següents etiquetes (Label) a les cel·les del GridPane.



Per a afegir una fila al **GridPane** selecciona un número de fila existent (es tornarà de color groc), fes clic dret sobre el número de fila i tria “Add Row”.



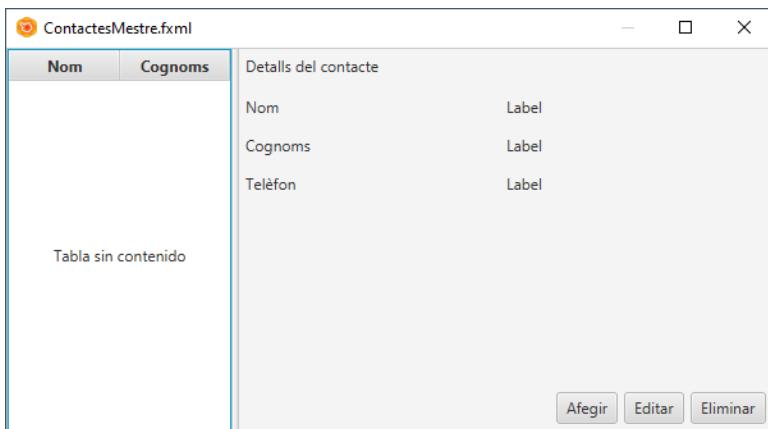
- 10 Afegeix 3 botons a la part inferior.



Selecciona'l sots, fes clic dret i invoca **Wrap In | HBox**. Això els posarà als 3 junts en un HBox. Podries necessitar establir un espaiat **spacing** dins del **HBox**.

Després, estableix també ancoratges (dret i inferior) perquè es mantinguin en el lloc correcte.

En aquest punt hauries de veure alguna cosa semblança al següent. Usa el menú Preview per a comprovar el seu comportament en canviar la grandària de la finestra.



60.1.5. Creació del formulari principal

Necessitem un altre arxiu FXML per a la nostra vista arrel, la qual contindrà una barra de menús i encapsularà la vista recentment creada **ContactesMestre.fxml**.

1. Crea un altre arxiu FXML dins del paquet vistes anomenat **LayoutPrincipal.fxml**.

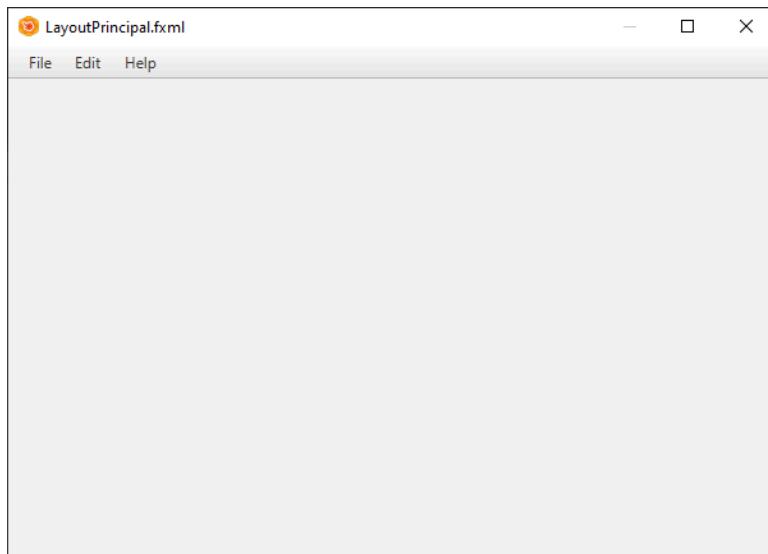
Modifica manualment el fitxer **FXML** generat per què l'element arrel sigui un **BorderPane**.

```
<BorderPane id="AnchorPane" prefHeight="400.0" prefWidth="600.0" xmlns:fx="http://javafx.com/fxml/1">

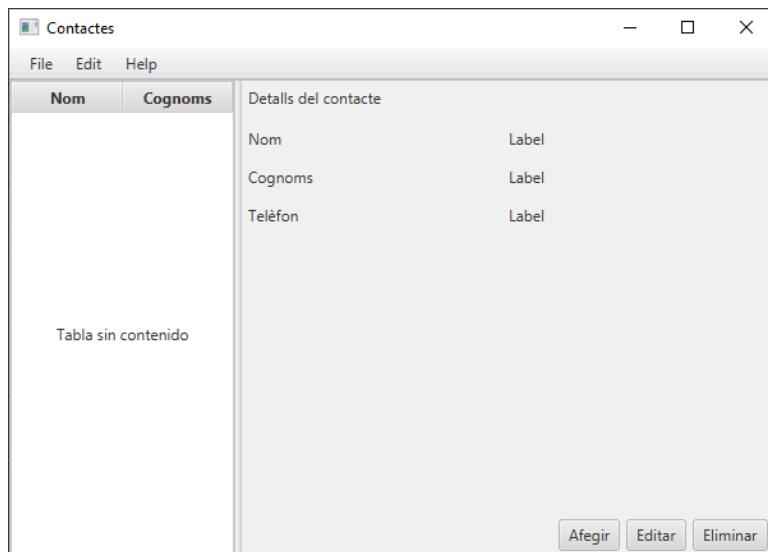
</BorderPane>
```

2. Canvia la grandària del BorderPane amb la propietat **prefWidth** establerta en 600 i **prefHeight** en 400.

3. Obre el formulari amb el Scene Builder i afegeix una **MenuBar** a la ranura superior del **BorderPane**. De moment no implementarem la funcionalitat del menú.



60.1.6. Implementació de la funció main



ContactesMain.java.

```
package contactes;

import java.io.IOException;
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.scene.layout.AnchorPane;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;

public class ContactesMain extends Application {

    private Stage primaryStage;
    private BorderPane root;

    @Override
    public void start(Stage primaryStage) {
        this.primaryStage = primaryStage;
        this.primaryStage.setTitle("Contactes");

        initRootLayout();

        mostrarContactesMestre();
    }

    public void initRootLayout() {
        try {
            FXMLLoader loader = new FXMLLoader();
            loader.setLocation(Contactes.class.getResource("vistes/
LayoutPrincipal.fxml"));
            root = (BorderPane) loader.load();

            Scene scene = new Scene(root);
            primaryStage.setScene(scene);
            primaryStage.show();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void mostrarContactesMestre() {
        try {
            FXMLLoader loader = new FXMLLoader();
```

```

        loader.setLocation(Contactes.class.getResource("vistes/
ContactesMestre.fxml"));
        AnchorPane personOverview = (AnchorPane) loader.load();

        root.setCenter(personOverview);
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public Stage getPrimaryStage() {
    return primaryStage;
}

public static void main(String[] args) {
    launch(args);
}
}

```

60.2. Implementació dels Model / Controlador

60.2.1. Model

Necessitem un model per contenir la informació sobre els contactes de la nostra agenda.

Afegeix una nova classe al paquet encarregat de contenir els models, **contactes.models** denominada **Contacte**. La classe **Contacte** tindrà atributs pel nom, cognoms i telèfon.

Com que voldrem vincular els diferents atributs dels diferents objectes **Contacte** als controls dels formularis caldrà que els atributs siguin capaços de detectar un canvi en un dels controls associats. Per fer-ho podem utilitzar **Properties** de JavaFX que encapsulen un tipus de dades bàsic i a més **són capaces de rebre notificacions quan el valor d'una variable canvia**.

Contacte.java.

```

package contactes.models;

import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;

```

```
public class Contacte {

    private final StringProperty nom;
    private final StringProperty cognoms;
    private final StringProperty telefon;

    public Contacte() {
        this(null, null);
    }

    public Contacte(String nom, String cognoms) {
        this.nom = new SimpleStringProperty(nom);
        this.cognoms = new SimpleStringProperty(cognoms);
        this.telefon = new SimpleStringProperty(null);
    }

    public String getNom() {
        return nom.get();
    }

    public void setNom(String nom) {
        this.nom.set(nom);
    }

    public StringProperty nomProperty() {
        return nom;
    }

    public String getCognoms() {
        return cognoms.get();
    }

    public void setCognoms(String cognoms) {
        this.cognoms.set(cognoms);
    }

    public StringProperty cognomsProperty() {
        return cognoms;
    }

    public String getTelefon() {
        return telefon.get();
    }

    public void setTelefon(String telefon) {
```

```

        this.telefon.set(telefon);
    }

    public StringProperty telefonProperty() {
        return telefon;
    }
}

```

60.2.2. L'origen de dades

Les principals dades que gestiona l' aplicació és una col·lecció de contactes. Crearem una llista d'objectes de tipus **Contacte** dins de la classe principal. La resta de controladors obtindrà després accés a aquesta col·lecció.

Com que volem que les dades dels contactes estiguin sincronitzades amb les vistes ens interessa algun tipus de col·lecció que sigui capaç d'informar a la vista dels canvis en les dades.

JavaFX proporciona la classe **ObservableList** amb aquesta capacitat.

ContactesMain.java.

```

package contactes;

import contactes.models.Contacte;
import java.io.IOException;
import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.scene.layout.AnchorPane;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;

public class ContactesMain extends Application {

    private Stage primaryStage;
    private BorderPane root;
    private ObservableList<Contacte> contactes =
        FXCollections.observableArrayList(); ①

    public ContactesMain() { ②

```

```
contactes.add(new Contacte("Elena", "Nito del Bosque"));
contactes.add(new Contacte("Elsa", "Capunta"));
contactes.add(new Contacte("Aitor", "Tilla"));
contactes.add(new Contacte("Débora", "Melo"));
contactes.add(new Contacte("Paca", "_Garte"));
contactes.add(new Contacte("Ana", "Tomía"));
contactes.add(new Contacte("Rubén", "Fermizo"));
contactes.add(new Contacte("Penélope", "Luda"));
contactes.add(new Contacte("Paul", "Vazo"));
contactes.add(new Contacte("Estela", "Gartija"));
}

public ObservableList<Contacte> getContactes() {❸
    return contactes;
}

@Override
public void start(Stage primaryStage) {
    this.primaryStage = primaryStage;
    this.primaryStage.setTitle("Contactes");

    initRootLayout();

    mostrarContactesMestre();
}

public void initRootLayout() {
    try {
        FXMLLoader loader = new FXMLLoader();
        loader.setLocation(ContactesMain.class.getResource("vistes/
LayoutPrincipal.fxml"));
        root = (BorderPane) loader.load();

        Scene scene = new Scene(root);
        primaryStage.setScene(scene);
        primaryStage.show();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public void mostrarContactesMestre() {
    try {
        FXMLLoader loader = new FXMLLoader();
```

```

        loader.setLocation(ContactesMain.class.getResource("vistes/
ContactesMestre.fxml"));
        AnchorPane personOverview = (AnchorPane) loader.load();

        root.setCenter(personOverview);
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public Stage getPrimaryStage() {
    return primaryStage;
}

public static void main(String[] args) {
    launch(args);
}
}

```

60.2.3. ContacteController

Per afegir dades a la taula ens farà falta controlador específic per a la vista **ContactesMestre.fxml**.

Crea una classe normal dins del paquet **vistes** denominat **ContactesMestreController.java**. (Hem de posar-ho en el mateix paquet que ContactesMestre.fxml o el Scene Builder no ho trobarà).

Afegirem alguns atributs per a accedir a la taula i les etiquetes de la vista. Aquests atributs aniran precedits per l'anotació **@FXML**. Això és necessari perquè la vista tingui accés als atributs i mètodes del controlador, fins i tot encara que siguin privats. Una vegada definida la vista en fxml, l'aplicació s'encarregarà d'emplenar automàticament aquests atributs en carregar el fxml.

ContactesMestreController.java.

```

package contactes.vistes;

import contactes.ContactesMain;
import contactes.models.Contacte;
import java.awt.Label;
import javafx.fxml.FXML;
import javafx.scene.control.TableColumn;

```

```
import javafx.scene.control.TableView;

public class ContactesMestreController {

    @FXML
    private TableView<Contacte> contacteTable;
    @FXML
    private TableColumn<Contacte, String> nomColumn;
    @FXML
    private TableColumn<Contacte, String> cognomsColumn;
    @FXML
    private Label nomLabel;
    @FXML
    private Label cognomsLabel;
    @FXML
    private Label telefonLabel;

    private ContactesMain main;

    @FXML
    private void initialize() {
        nomColumn.setCellValueFactory(cellData ->
cellData.getValue().nomProperty());
        cognomsColumn.setCellValueFactory(cellData ->
cellData.getValue().cognomsProperty());
    }

    public void setMainApp(ContactesMain main) {
        this.main = main;

        contacteTable.setItems(main.getContactes());
    }
}
```

Aquest codi necessita certa explicació:

Els camps i mètodes on l'arxiu fxml necessita accés han de ser anotats amb **@FXML**. En realitat, només si són privats, però és millor tenir-los privats i marcar-los amb l'anotació.

El mètode **initialize()** és invocat automàticament després de carregar el fxml. En aquest moment, tots els atributs FXML ja haurien haver estat inicialitzats..

El mètode **setCellValueFactory(...)** que apliquem sobre les columnes de la taula s'usa per a determinar quins atributs de la classe **Contacte** han de ser usats per a cada columna particular.

60.2.4. Enllaçar l'aplicació principal amb ContactesMestreController

El mètode **setMainApp(...)** ha de ser invocat des de la classe **ContactesMain**. Això ens dóna l'oportunitat d'accedir a l'objecte **ContactesMain** per a obtenir la llista de contactes i altres coses. Substitueix el mètode **mostrarContactesMestre()** amb el codi següent, el qual conté dues línies addicionals:

ContactesMain.java.

```
package contactes;

import contactes.models.Contacte;
import contactes.vistes.ContactesMestreController;
import java.io.IOException;
import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.scene.layout.AnchorPane;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;

public class ContactesMain extends Application {

    private Stage primaryStage;
    private BorderPane root;
    private ObservableList<Contacte> contactes =
    FXCollections.observableArrayList();

    public ContactesMain() {
        contactes.add(new Contacte("Elena", "Nito del Bosque"));
        contactes.add(new Contacte("Elsa", "Capunta"));
        contactes.add(new Contacte("Aitor", "Tilla"));
        contactes.add(new Contacte("Débora", "Melo"));
        contactes.add(new Contacte("Paca", "_Garte"));
        contactes.add(new Contacte("Ana", "Tomia"));
    }
}
```

Enllaçar l'aplicació principal
amb ContactesMestreController

```
contactes.add(new Contacte("Rubén", "Fermizo"));
contactes.add(new Contacte("Penélope", "Luda"));
contactes.add(new Contacte("Paul", "Vazo"));
contactes.add(new Contacte("Estela", "Gartija"));

}

public ObservableList<Contacte> getContactes() {
    return contactes;
}

@Override
public void start(Stage primaryStage) {
    this.primaryStage = primaryStage;
    this.primaryStage.setTitle("Contactes");

    initRootLayout();

    mostrarContactesMestre();
}

public void initRootLayout() {
    try {
        FXMLLoader loader = new FXMLLoader();
        loader.setLocation(ContactesMain.class.getResource("vistes/
LayoutPrincipal.fxml"));
        root = (BorderPane) loader.load();

        Scene scene = new Scene(root);
        primaryStage.setScene(scene);
        primaryStage.show();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public void mostrarContactesMestre() {
    try {
        FXMLLoader loader = new FXMLLoader();
        loader.setLocation(ContactesMain.class.getResource("vistes/
ContactesMestre.fxml"));
        AnchorPane personOverview = (AnchorPane) loader.load();

        root.setCenter(personOverview);
    }
```

```
    ContactesMestreController controller =
loader.getController(); ❶
    controller.setMainApp(this); ❷

} catch (IOException e) {
    e.printStackTrace();
}

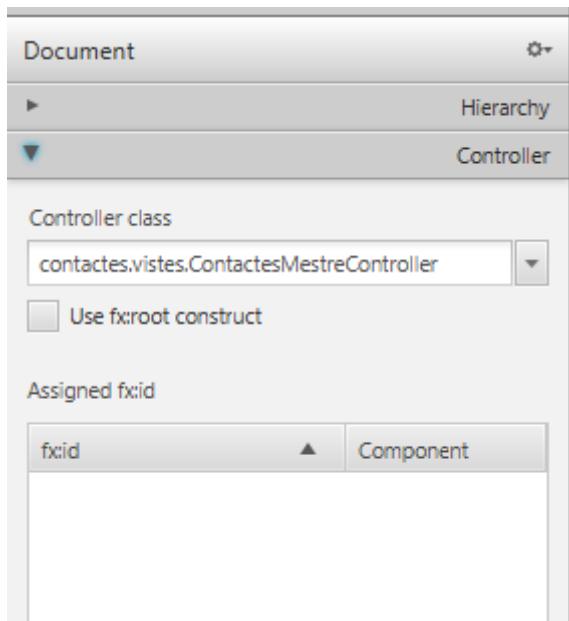
public Stage getPrimaryStage() {
    return primaryStage;
}

public static void main(String[] args) {
    launch(args);
}
}
```

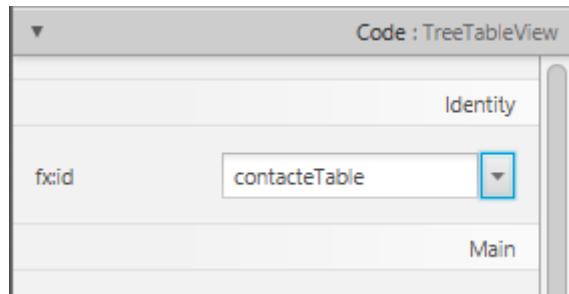
60.2.5. Vincular la vista al controlador

Ja gairebé el tenim! Però falta un detall: no li hem indicat a la vista declarada a **ContactesMestre.fxml** quin és el seu controlador i que element fer corresponer a cada un dels atributs en el controlador.

1. Obre **ContactesMestre.fxml** en SceneBuilder.
2. Obre la secció Controller en el costat esquerre i selecciona **ContactesMestreController** com a controlador.



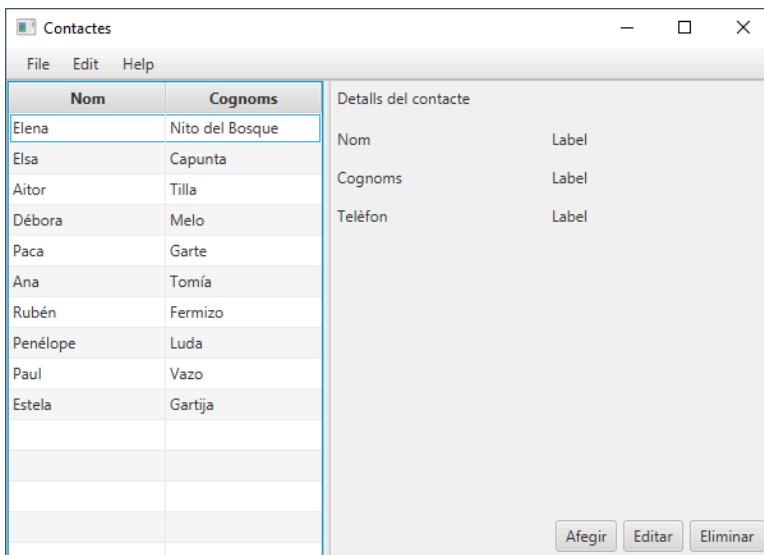
3. Selecciona TableView en la secció Hierarchy i en l'apartat Code escriu **contacteTable** en la propietat **fx:id**.



4. Fes el mateix per a les columnes, posant **nomColumn** i **cognomsColumn** com les seves **fx:id** respectivament.
5. Per a cada etiqueta en la segona columna, introduceix el **fx:id** que correspongui.



Arribats a aquest punt l'aplicació hauria de complir i mostrar una cosa similar a la següent captura:



60.3. Esdeveniments

60.3.1. Resposta a canvis en la selecció de la Taula

Encara no hem usat la part dreta de la interfície de la nostra aplicació. La intenció és usar aquesta part per a mostrar els detalls de la persona seleccionada per l'usuari en la taula.

En primer lloc afegirem un nou mètode dins de **ContactesMestreController** que ens ajudi a emplenar les etiquetes amb les dades d'un contacte..

1. Crea un mètode anomenat **mostrarDetailsContacte(Contacte contacto)**. Aquest mètode recorrerà totes les etiquetes i establirà el text amb detalls del

contacte usant **setText(...)**. Si en comptes d'una instància de **Contacte** es passa **null** llavors les etiquetes han de ser esborrades.

ContactesMestreController.java.

```
private void mostrarDetallsContacte(Contacte contacte) {
    if (contacte != null) {
        nomLabel.setText(contacte.getNom());
        cognomsLabel.setText(contacte.getCognoms());
        telefonLabel.setText(contacte.getTelefon());
    } else {
        nomLabel.setText("");
        cognomsLabel.setText("");
        telefonLabel.setText("");
    }
}
```

60.3.2. Detecta canvis de selecció en la taula

Per a assabentar-se que l'usuari ha seleccionat a un contacte a la taula de contactes, necessitem escoltar els canvis. Això s'aconsegueix mitjançant la implementació d'una interfície de JavaFX que es diu **ChangeListener**, aquesta interfície té un sol mètode anomenat **changed(...)**. Aquest mètode només té tres paràmetres: **observable**, **oldValue**, i **newValue**.

1. Afegirem algunes línies al mètode **initialize()** de **ContactesMestreController**. El codi resultant s'assemblarà al següent:

ContactesMestreController.java.

```
@FXML
private void initialize() {
    nomColumn.setCellValueFactory(cellData ->
    cellData.getValue().nomProperty());
    cognomsColumn.setCellValueFactory(cellData ->
    cellData.getValue().cognomsProperty());

    mostrarDetallsContacte(null); ①

    contacteTable.getSelectionModel().selectedItemProperty().addListener(
```

```

        (observable, oldValue, newValue) ->
    mostrarDetallsContacte(newValue)); ②
}

```

- ①** Amb **mostrarDetallsContacte(null)**; esborrem els detalls d'un contacte.
- ②** Amb **contacteTable.getSelectionModel()**... obtenim la **selectedItemProperty** de la taula de contactes, i li afegim un "listener". Quan un usuari seleccioni a una persona a la taula, l'expressió lambda serà executada: es pren la persona seleccionada i se li passa al mètode **mostrarDetallsContacte(...)**.

Si executem l'aplicació en aquest moment al seleccionar un contacte, els detalls d'aquests són mostrats a la part dreta de la finestra.

60.3.3. El botó d'esborrar

La nostra interfície d'usuari ja conté un botó d'esborrar, però sense funcionalitat. Podem seleccionar l'acció a executar en premer un botó des del Scene Builder. Qualsevol mètode del nostre controlador anotat amb @FXML (o declarat com public) és accessible des de Scene Builder. Així doncs,

1. Comencem afegint el mètode d'esborrat al final de nostra **ContactesMestreController**:

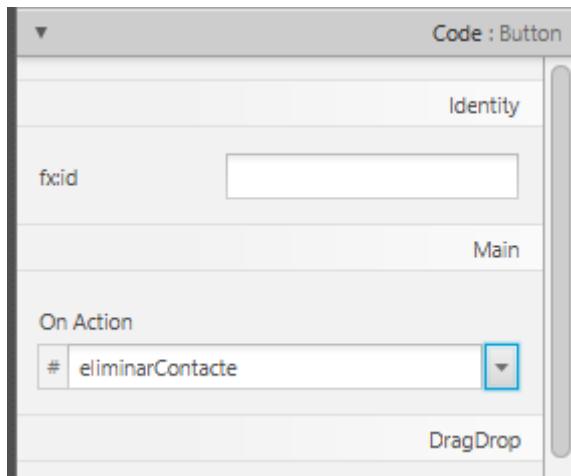
ContactesMestreController.java.

```

@FXML
private void eliminarContacte() {
    int selectedIndex =
    contactoTable.getSelectionModel().getSelectedIndex();
    contactoTable.getItems().remove(selectedIndex);
}

```

2. Ara, obrim l'arxiu **ContactesMestre.fxml** en el SceneBuilder. Selecciona el botó Eliminar, obre l'apartat Code i posa **eliminarContacte()** en el menú desplegable denominat **On Action**.



60.3.4. Gestió d'errors

Si executes l'aplicació en aquest punt hauries de ser capaç d'esborrar contactes de la taula. Però, què ocorre si es prem el botó d'esborrar sense seleccionar a ningú en la taula.

Es produeix un error de tipus **ArrayIndexOutOfBoundsException** perquè no pot esborrar un contacte en l'índex -1, que és el valor retornat pel mètode **getSelectedIndex()** quan no hi ha cap element seleccionat.

1. Per evitar-ho modifiquem el mètode **eliminarContacte()** per mostrar l'error a l'usuari.

ContactesMestreController.java.

```
@FXML
private void eliminarContacte() {
    int selectedIndex =
        contacteTable.getSelectionModel().getSelectedIndex();
    if (selectedIndex >= 0) {
        contacteTable.getItems().remove(selectedIndex);
    } else {
        Alert alert = new Alert(Alert.AlertType.WARNING);
        alert.setTitle("Contacte no seleccionat");
        alert.setHeaderText("No s'ha seleccionat cap contacte");
        alert.setContentText("Cal seleccionar un contepte abans
d'eliminar-lo");
```

```

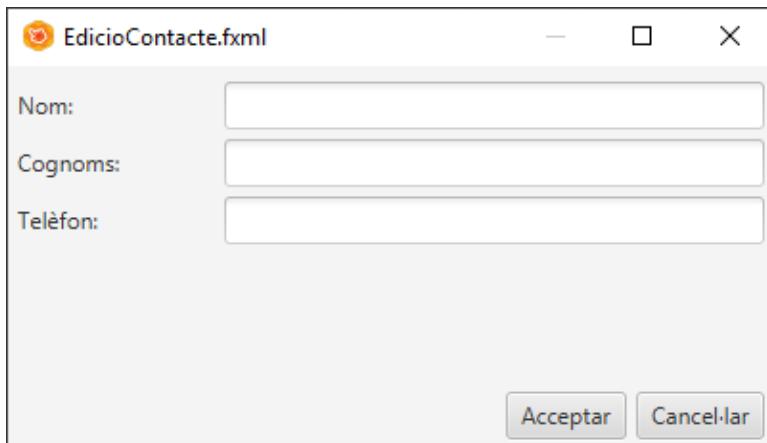
        alert.showAndWait();
    }
}

```

60.3.5. Diàlegs per a crear i editar contactes

Les accions d'editar i crear nou contacte necessiten una mica més d'elaboració: necessitarem una finestra de diàleg a mesura (és a dir, un nou **Stage**) amb un formulari per a preguntar a l'usuari els detalls sobre la persona. Dissenya la finestra de diàleg

1. Crea un nou arxiu **fxml** anomenat **PersonEditDialog.fxml** dins del paquet **view**.
2. Usa un panell de reixa (**GridPane**), etiquetes (**Label**), camps de text (**TextField**) i botons (**Button**) per a crear una finestra de diàleg com la següent:



3. Crea el controlador per a la finestra d'edició de persones i anomena'l **PersonEditDialogController.java**:

```

package contactes.vistes;

import contactes.models.Contacte;
import javafx.fxml.FXML;
import javafx.scene.control.Alert;
import javafx.scene.control.TextField;

```

```
import javafx.stage.Stage;

public class EdicioContacteController {

    @FXML
    private TextField nomField;
    @FXML
    private TextField cognomField;
    @FXML
    private TextField telefonField;

    private Stage dialogStage;
    private Contacte contacte;
    private boolean okClicked = false;

    @FXML
    private void initialize() {
    }

    public void setDialogStage(Stage dialogStage) {
        this.dialogStage = dialogStage;
    }

    public void setContacte(Contacte contacte) {
        this.contacte = contacte;

        nomField.setText(contacte.getNom());
        cognomField.setText(contacte.getCognoms());
        telefonField.setText(contacte.getTelefon());
    }

    public boolean acceptarClicked() {
        return okClicked;
    }

    @FXML
    private void acceptar() {
        if (isValidInput()) {
            contacte.setNom(nomField.getText());
            contacte.setCognoms(cognomField.getText());
            contacte.setTelefon(telefonField.getText());

            okClicked = true;
            dialogStage.close();
        }
    }
}
```

```
}

@FXML
private void cancelar() {
    dialogStage.close();
}

private boolean isInputValid() {
    String errorMessage = "";

    if (nomField.getText() == null || nomField.getText().length() == 0) {
        errorMessage += "Nom no vàlid\n";
    }
    if (cognomField.getText() == null || cognomField.getText().length() == 0) {
        errorMessage += "Cognoms no vàlids!\n";
    }
    if (telefonField.getText() == null || telefonField.getText().length() == 0) {
        errorMessage += "Telèfon no vàlid!\n";
    }

    if (errorMessage.length() == 0) {
        return true;
    } else {
        Alert alert = new Alert(Alert.AlertType.ERROR);
        alert.setTitle("Camps no vàlids");
        alert.setHeaderText("Si et plau corregeix els camps invàlids");
        alert.setContentText(errorMessage);

        alert.showAndWait();

        return false;
    }
}
```

60.3.6. Enllaçar la vista i el controlador

Una vegada creades la vista (FXML) i el controlador, necessitem vincular l'u amb l'altre:

1. Obre l'arxiu **EdicioContacte.fxml**.
2. En la secció Controller a l'esquerra selecciona **EdicioContacteController** com a classe de control.
3. Estableix el camp **fx:id** de totes els *TextField* amb els identificadors dels atributs del controlador corresponents.
4. Especifica el camp **onAction** dels dos botons amb els mètodes del controlador corresponents a cada acció.

60.3.7. Obrint la finestra de diàleg

1. Afegeix un mètode per a carregar i mostrar el mètode d'edició d'una persona dins de la classe MainApp:

ContactesMain.java.

```
public boolean mostrarEdicioContacte(Contacte contacte) {
    try {
        FXMLLoader loader = new FXMLLoader();
        loader.setLocation(ContactesMain.class.getResource("vistes/
EdicioContacte.fxml"));
        AnchorPane page = (AnchorPane) loader.load();

        Stage dialogStage = new Stage();
        dialogStage.setTitle("Editar Contacte");
        dialogStage.initModality(Modality.WINDOW_MODAL);
        dialogStage.initOwner(primaryStage);
        Scene scene = new Scene(page);
        dialogStage.setScene(scene);

        EdicioContacteController controller =
        loader.getController();
        controller.setDialogStage(dialogStage);
        controller.setContacte(contacte);

        dialogStage.showAndWait();

        return controller.acceptarClicked();
    } catch (IOException e) {
        e.printStackTrace();
        return false;
    }
}
```

```
}
```

2. Afegeix els següents mètodes a la classe **EdicioContacteController**. Aquests mètodes cridaran al mètode **mostrarEdicioContacte(...)** des de **ContactesMain** quan l'usuari premi en els botons **afegir** o **editar**.

ContactesMestreController.java.

```

@FXML
private void afegirContacte() {
    Contacte nouContacte = new Contacte();
    boolean acceptarClicked =
main.mostrarEdicioContacte(nouContacte);
    if (acceptarClicked) {
        main.getContactes().add(nouContacte);
    }
}

@FXML
private void editarContacte() {
    Contacte contacteSeleccionat =
contacteTable.getSelectionModel().getSelectedItem();
    if (contacteSeleccionat != null) {
        boolean okClicked =
main.mostrarEdicioContacte(contacteSeleccionat);
        if (okClicked) {
            mostrarDetallsContacte(contacteSeleccionat);
        }
    } else {

        Alert alert = new Alert(Alert.AlertType.WARNING);
        alert.setTitle("Sel·lecció incorrecta");
        alert.setHeaderText("No s'ha sel·leccionat cap contacte
de la taula");
        alert.setContentText("Sel·lecciona un contacte de la
taula.");
        alert.showAndWait();
    }
}

```

3. Obre l'arxiu **ContactesMestre.fxml** mitjançant Scene Builder. Tria els mètodes corresponents en el camp **On Action** per als botons **afegir** i **editar**.

61

Entrada / Sortida

Els processos d'entrada/sortida treballen llegint dades d'un **origen** i escrivint dades a un **destí**.

L'origen i el destí **no són necessàriament fitxers**, per exemple, el teclat opera com l'entrada estàndard encapsulada en Java per l'objecte **System.in** i la pantalla opera com la sortida estàndard, encapsulada en Java dins de l'objecte **System.out**.

Tant l'entrada com la sortida estàndard es poden redireccionar a d'altres orígens i a d'altres destins.

61.1. Paquets **java.io** i **java.nio**

Els paquets **java.io** i **java.nio** contenen les classes que gestionen l'entrada i la sortida. Les classes del **java.nio** gestionen la entrada i la sortida de dades amb **buffers** a diferència de les classes del paquet **java.io** que treballen directament amb **streams o fluxos**.

En general les classes del paquet **java.io** són més senzilles d'utilitzar i són amb les que treballarem en aquest apartat, no obstant les diferències principals entre els dos paquets són les següents:

java.io	java.nio
Treballa amb "Streams"	Treballa amb "buffers"
Les operacions E/S bloquegen	Les operacions E/S no bloquegen
	Treballa amb Selectors

Treballar amb "Streams" vs treballar amb "buffers"

- Treballar amb "streams" significa que els bytes es llegeixen directament d'un flux de dades, la gestió posterior d'aquests bytes només depèn del programa, els bytes no s'emmagatzemem a cap cache. Per tant, no és possible, d'entrada, moure's endavant i endarrere pel flux de bytes.
- Quan es treballa amb "buffers" les dades no es llegeixen directament del flux de dades sinó que es guarden en un "buffer" i es processen des d'allà. Això permet moure's endavant i endarrere pel "buffer" segons convingui. No obstant, cal comprovar si les dades necessàries estan o no al "buffer" cosa que en complica la gestió.

Operacions E/S que bloquegen vs operacions E/S que no bloquegen

- **java.io** bloqueja el fil d'execució actual, això vol dir que quan es realitza una operació de lectura o d'escriptura l'execució del codi s'interromp en espera de la lectura o escriptura de les dades.
- Les classes de **java.nio** no bloquegen el fil d'execució quan realitzen operacions de lectura o escriptura, per tant es poden demanar dades d'un canal, obtenir només el que estigui disponible en aquell moment i seguir treballant en alguna altra cosa mentre s'esperen noves dades.

Selectors

Els selectors de **java.nio** permeten a un sol fil d'execució monitorar més d'un canal E/S i seleccionar dinàmicament quins d'ells tenen dades per processar.

61.2. Fluxos basats en bytes i fluxos basats en text

Java defineix dos tipus diferents de fluxos, fluxos basats en bytes i fluxos basats en text.

Els **fluxos basats en bytes** treballen directament amb els bytes i són convenient per treballar amb dades binàries.

Els **fluxos basats en caràcters** utilitzen unicode i per tant poden ser internacionalitzats.

Aquest tractament diferenciat genera classes diferenciades pel tractament de fluxos de caràcters i de bytes i com a conseqüència la quantitat de classes dedicades al tractament de fluxos en Java **és molt elevada**.

No obstant, cal tenir present que en general les classes que treballen amb fluxos de bytes solen tenir una classe de funcionament equivalent pels fluxos de caràcters, fet que simplifica la interpretació correcte de les biblioteques dedicades a fluxos de Java.

62

Lectura i escriptura de fitxers

Essencialment hi ha dues maneres d'**interpretar** dades emmagatzemades, en **format text** o en **format binari**.

- En **format text** les dades es s'interpreten com una seqüència de caràcters. A més, aquesta interpretació és susceptible de la codificació de caràcters emprada.
- En **format binari** les dades s'interpreten com a una seqüència de **bytes**.

Per exemple:

- en format text i codificació ASCII, la cadena **àabc** s'emmagatzemaria com la seqüència de bits,

```
11100000 01100001 01100010 01100011
```

- però la mateixa cadena en codificació UTF-8, s'emmagatzemaria amb una seqüència de bits diferent,

```
11000011 10100000 01100001 01100010 01100011
```

Si llegim les dades de l'exemple com a text obtindrem en ambdós casos la mateixa cadena, **àabc**. En canvi si llegim les dades en format binari obtindrem diferents seqüències de bits.

62.1. Classes de la biblioteca de Java

La biblioteca de Java proporciona dos conjunts de classes per gestionar la entrada i la sortida.

Classes **xxxStream**

La família de classes **xxxStream** gestionen les dades d'un origen de dades de forma **binària**, és a dir, byte a byte.

Classes **xxxReader** i **xxxWriter**

La família de classes **xxxWriter** i **xxReader** gestionen les dades en format text, és a dir, caràcter a caràcter.

62.2. Per què calen dos conjunts de classes?

Tot i que els caràcters estan formats per bytes existeixen variacions respecte com es representa cada caràcter. Per exemple, el caràcter 'é' es codifica com un únic bit de valor 223 en la codificació ISO-8859-1, típicament utilitzada a Nord Amèrica i a Europa occidental. En canvi en la codificació UTF-8 el caràcter 'é' es codifica amb dos bytes, 195 i 169, i en la codificació UTF-16 es codificantaria com a 0 223.

Les classes **Reader** i **Writer** tenen la responsabilitat de convertir entre bytes i caràcters, tenint en compte les diferents codificacions de caràcters involucrades.

Per defecte utilitzen la configuració de caràcters especificada pel sistema operatiu que executa el programa, no obstant, es pot especificar una codificació diferent a l'hora de construir l'objecte Reader o Writer. Per exemple:

```
Scanner in = new Scanner(input, "UTF-8");
PrintWriter out = new PrintWriter(output, "UTF-8");
```



No hi ha manera de determinar automàticament la codificació de caràcters utilitzada en un text particular. Cal conèixer quina codificació es va utilitzar en el moment de la escriptura del text i adaptar les classes de lectura a aquesta codificació.

62.3. Tractament seqüencial o aleatori

A part de tenir classes per tractar fitxers en base al seu contingut també es proporcionen diferents classes per tractar fitxers en base a la manera d'accendir-hi.

Fitxers d'accés seqüencial

Com que els valors d'un fitxer es solen emmagatzemar en forma de seqüència, un rere l'altre, la manera més habitual de tractar fitxers és seqüencialment.

S'anomena **accés seqüencial** al tractament d'un conjunt d'elements de manera que només és possible accedir-hi d'acord al seu l'ordre d'aparició i per tant, per a poder tractar un element, cal haver tractat tots els elements anteriors.

Fitxers d'accés aleatori o accés directe

L'accés aleatori fa referència a la habilitat d'accendir a una dada en unes coordenades donades dins d'un conjunt d'elements adreçables.

62.4. Java IO: Streams

Els següents diagrames mostren part de la jerarquia de classes que treballen amb fluxos de dades, és a dir, aquestes classes **accedeixen a les dades com a un a seqüència de bytes**. En general les dades s'extrauran o s'enviaran a connexions de xarxa, fitxers o dispositius externs.

En funció de les nostres necessitats s'escolllirà una de les classes o una altra, per exemple, la solució més senzilla per treballar amb orígens i destins de tipus fitxer serà escolllir les classes **FileInputStream** i **FileOutputStream**.

Es mostren dues jerarquies de classes, en primer lloc les classes que s'utilitzen per **llegir** dades seqüencialment i en format binari i en segon lloc les classes que s'utilitzen per **escriure** dades seqüencialment en format binari.

Quan parlem de **jerarquia** fem referència a que les classes situades més avall, a la cua de les fletxes, són **especialitzacions** de les situades més amunt, per exemple, en el diagrama següent estem dient que un **PrintStream** és en particular un **FilterOutputStream** i aquest és en particular un **OutputStream**.

Com a conseqüència un **PrintStream** tindrà tots els mètodes que té un **FilterOutputStream** i possiblement més, i un **FilterOutputStream** tindrà tots els mètodes que té un **OutputStream** i possiblement més.

En general, aquestes classes no mantenen un index per llegir o escriure les dades i en general no permet la lectura o escriptura endavant o endarrere del stream.

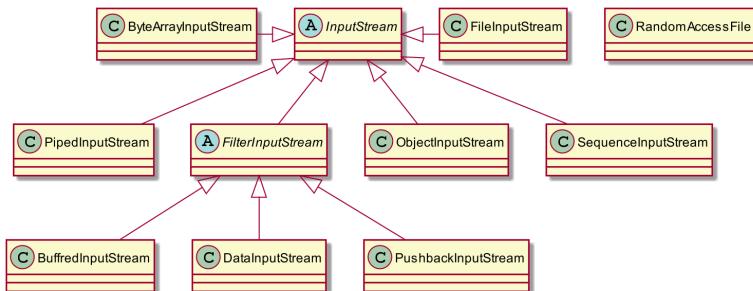


Figura 62.1. Classes habituals de lectura de "streams" binaris ¹

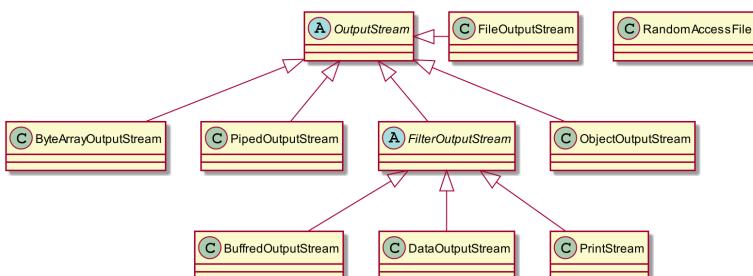


Figura 62.2. Classes habituals d'escriptura de "streams" binaris ²

BuffredInputStream	Realitza la lectura/ escriptura contra un buffer enlloc de directament contra les dades.
BufferedOutputStream	
ByteArrayInputStream	Llegeix/escriu a una matriu de bytes
ByteArrayOutputStream	
DataInputStream	Disposa de mètodes per llegir/escriure els tipus de dades estàndard de Java
DataOutputStream	
FileInputStream	Llegeix/escriu d'un fitxer

¹ <https://docs.oracle.com/javase/8/docs/api/java/io/InputStream.html>

² <https://docs.oracle.com/javase/8/docs/api/java/io/OutputStream.html>

FileOutputStream	
ObjectInputStream	
ObjectOutputStream	Disposa de mètodes per llegir/escriure objectes
PipedInputStream	S'utilitza per connectar streams de dades entre diferents fils d'execució.
PipedOutputStream	
PushbackInputStream	InputStream que permet retornar bytes al stream
PrintStream	OutputStream que disposa dels mètodes print() i println()
SequenceInputStream (JDK11)	InputStream que és combinació de dos o més streams que es llegiran seqüencialment, un darrera l'altre.

62.5. Java IO: Readers i Writers

Els següents diagrames mostren part de la jerarquia de classes que treballen amb "readers" i "writers", és a dir, aquestes classes **accedeixen a les dades com a un a seqüència de caràcters**.

En funció de les nostres necessitats s'escolllirà una de les classes o una altra, per exemple, la solució més senzilla per treballar amb orígens i destins de tipus fitxer serà escollir les classes **FileReader** i **FileWriter**.

Es mostren dues jerarquies de classes, en primer lloc les classes que s'utilitzen per **llegir** dades seqüencialment i en format caràcter i en segon lloc les classes que s'utilitzen per **escriure** dades seqüencialment en format caràcter..

Les classe **java.io.Reader** i la classe **java.io.Writer** funcionen de forma similar a les classes InputStream i OutputStream amb la diferència que les primeres són basades en **caràcters** i es segones són basades en bytes. Estan pensades per **treballar amb text**.

En general, aquestes classes no mantenen un index per llegir o escriure les dades i en general no permet la lectura o escriptura endavant o endarrere del stream.

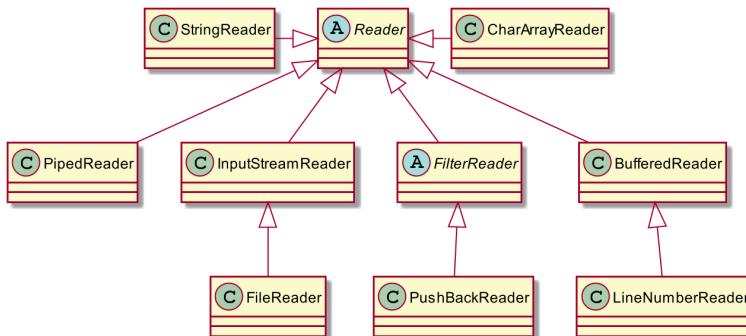


Figura 62.3. Part de la jerarquia de classes orientades a la lectura de fitxers de text ³

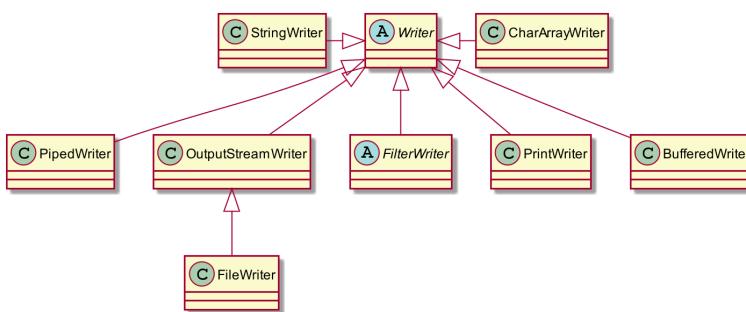


Figura 62.4. Part de la jerarquia de classes orientades a l'escriptura de fitxers de text ⁴

BuffredReader	Realitza la lectura/ escriptura contra un buffer enlloc de directament contra les dades.
BufferedWriter	
CharArrayReader	
CharArrayWriter	Llegeix/escriu a una matriu de caràcters
FileWriter	
FileReader	Llegeix/escriu d'un fitxer
LineNumberReader (JDK11)	BufferedInputReader que compta línies
OutputStreamWriter	OutputStream capaç de transformar caràcters en bytes

³ <https://docs.oracle.com/javase/8/docs/api/java/io/Reader.html>

⁴ <https://docs.oracle.com/javase/8/docs/api/java/io/Writer.html>

PipedReader	S'utilitza per connectar streams de dades entre diferents fils d'execució.
PrintWriter	OutputStream que disposa dels mètodes print() i println()
PushBackReader	InputStream que permet retornar caràcters al stream
StringReader	Permet llegir/escriure des d'un String
StringWriter	

63

Treballar amb fitxers

63.1. Conèixer el directori de treball

El concepte de **directorí de treball** està relacionat amb els sistemes operatius, no és un concepte de Java.

Quan un procés comença, utilitza el seu directori de treball per resoldre les rutes relatives a fitxers, quan s'executa un programa Java, la màquina virtual Java s'executa com un procés i com a conseqüència té un directori de treball actual. El valor del directori de treball actual per una màquina virtual Java depèn de com s'ha executat la comanda **java**.

Conèixer quin és el directori de treball quan s'està programant amb fitxers és molt important ja que determina quin és el directori arrel per a les rutes relatives utilitzades en el programa.

Per obtenir el directori de treball actual cal llegir la propietat del sistema **user.dir**.

```
String workingDir = System.getProperty("user.dir");
```

També es pot especificar el directori de treball actual que utilitzarà la màquina virtual de Java al arrencar:

```
java -Duser.dir=C:\nou-working-dir la-meva-classe
```

63.2. "Paths" o rutes

Cada fitxer es troba dins d'un directori dins el sistema de fitxers, ens podem referir als fitxers a través d'una ruta.



Una **ruta**, amb anglès **path** és la forma general d'un nom de fitxer o carpeta, de manera que identifica la seva localització en el sistema de fitxers.

Les rutes poden ser de dos tipus:

ruta absoluta d'un fitxer

És aquella que es refereix a un element destí a partir de la carpeta arrel d'un sistema de fitxers. En aquesta s'enllaren, una per una, totes les carpetes que hi ha entre la carpeta arrel fins a la carpeta que conté l'element destinació.

Per exemple, **c:\javaprojects\Exercici.java** o bé **/etc/network/interfaces**.

ruta relativa d'un fitxer

És aquella que es considera que parteix des del **directorí de treball** de l'aplicació. Aquesta carpeta pot ser diferent cada cop que s'executa el programa i depèn de la manera com s'ha dut a terme aquesta execució.

Per exemple, si el directori de treball actual és **c:\javaprojects**, **Exercici.java** és la ruta relativa al fitxer **c:\javaprojects\Exercici.java**.



Cal tenir sempre present si el sistema operatiu distingeix entre majúscules i minúscules o no.



El format de la cadena de text que conforma la ruta pot ser diferent segons el sistema operatiu sobre el qual s'executa l'aplicació.

Per exemple, el sistema operatiu Windows inicia les rutes per un nom d'unitat (C:, D:, etc.), mentre que els sistemes operatius basats en Unix comencen directament amb una barra ("").

A més a més, els diferents sistemes operatius usen diferents separadors dins les rutes.

Per exemple, els sistemes Unix usen la barra ("/") mentre que el Windows la contrabarra ("\").

63.3. La classe File

Tot i que la majoria de classes definides en el paquet **java.io** operen sobre streams, la classe **File** no. Opera directament amb els fitxers i el sistema de fitxers subjacent. Això vol dir que la classe **File** **no especifica com es recupera o s'emmagatzema** informació en un fitxer, només **en descriu les propietats**.

La classe **File** és una **representació abstracte dels noms de les rutes a carpetes i fitxers**. Conté mètodes per **obtenir les propietats** d'un fitxer o d'un directori i per **renombrar i eliminar** un fitxer o un directori.



La interfície **Path** i la classe **Files** de **java.nio** proporcionen una bona alternativa a la classe **File**.

java.io

C File

- `File(pathname: String)`
- `File(parent: String, child: String)`
- `File(parent: File, child: String)`
- `exists(): boolean`
- `canRead(): boolean`
- `canWrite(): boolean`
- `createNewFile(): boolean`
- `isDirectory(): boolean`
- `isFile(): boolean`
- `isAbsolute(): boolean`
- `isHidden(): boolean`
- `getAbsolutePath(): String`
- `getCanonicalPath(): String`
- `getName(): String`
- `getPath(): String`
- `getParent(): String`
- `lastModified(): long`
- `length(): long`
- `listFiles(): File[]`
- `listRoots(): File[]`
- `delete(): boolean`
- `renameTo(dest: File): boolean`
- `setReadonly(): boolean`
- `setReadable(boolean readable): boolean`
- `setWritable(boolean writable): boolean`
- `setExecutable(boolean executable): boolean`
- `mkdir(): boolean`
- `mkdirs(): boolean`

File(pathname: String)

Crea un objecte File per la ruta especificada, **pathname** pot ser un fitxer o un directori.

```
File f1 = new File("//");
```

File(parent: String, child: String)

Crea un objecte File per **child** dins del directori parent, **child** pot ser un fitxer o un directori.

```
File f2 = new File("/", "autoexec.bat");
```

File(parent: File, child: String)

Crea un objecte File per child dins del directori parent. parent és un objecte File.

```
File f3 = new File(f1, "autoexec.bat");
```

exists(): boolean

Retorna true si el fitxer o directori representat per l'objecte File existeix. El mètode **exists** comprova si el path assignat a un objecte File existeix.

canRead: boolean

Retorna true si el fitxer especificat existeix i té permisos de lectura per l'aplicació.

canWrite: boolean

Retorna true si el fitxer especificat d'escriptura per l'aplicació.

createNewFile(): boolean

Crea un fitxer nou, buit si el nom especificat a l'objecte File no existeix.

isDirectory

Retorna true si el fitxer indicat per la ruta és un directori.

isFile()

Retorna true si el fitxer indicat per la ruta és un fitxer.

isAbsolute()

Retorna true si el fitxer indicat per la ruta és una ruta absoluta.

isHidden()

Retorna true si el fitxer indicat per la ruta és està marcat com a ocult.

getAbsolutePath(): String

Retorna la ruta sencera del fitxer o directori representat per l'objecte File.

getCanonicalPath(): String

Retorna el mateix que el mètode **getAbsolutePath** però elimina tota la informació redundant de la ruta.

Elimina els . i .., resol els links simbòlics, en Linux, i posa la lletra de la unitat en majúscula, en Windows.

getName(): String

Retorna el nom del fitxer o del directori indicat pel path.

getPath(): String

Retorna la ruta encapsulada al objecte File en una cadena.

getParent(): String

Retorna la ruta del directori pare del fitxer. Retorna null en cas que el fitxer no tingui un directori pare.

lastModified(): long

Retorna el temps de l'última modificació del fitxer.

length(): long

Retorna la mida del fitxer en bytes.

listfiles(): File[]

Retorna una matriu de rutes indicant els fitxers del directori indicat pel path.

listRoots():File[]

Retorna una matriu indicant els diferents directoris arrel del sistema de fitxers.

delete(): boolean

Elimina el fitxer o directori representat per l'objecte File. Retorna **true** si l'eliminació té èxit.

renameTo(dest: File): boolean

Renombra a **dest** el fitxer o directori representat per l'objecte File. Retorna **true** si l'operació té èxit.

setReadonly(): boolean

Marca el fitxer o el directori com a només lectura.

setReadable(boolean readable): boolean

Proporciona permisos de lectura pel propietari del fitxer. Retorna **true** si l'operació ha tingut èxit.

setWritable(boolean writable): boolean

Proporciona permisos d'escritura pel propietari del fitxer. Retorna **true** si l'operació ha tingut èxit.

setExecutable(boolean executable): boolean

Proporciona permisos d'execució pel propietari del fitxer. Retorna **true** si l'operació ha tingut èxit.

mkdir(): boolean

Crea el directori representat per l'objecte File. Retorna **true** si el directori es crea amb èxit.

mkdirs(): boolean

Funciona igual que **mkdir()** excepte que crea el directori junt amb els directoris pare si aquests no existeixen.

63.4. Creació d'un objecte File

Es pot crear un objecte File a partir d'una ruta relativa o absoluta.

Per exemple:

```
File fitxerTest = new File("fitxerTest.txt");
```



El caràcter (/) és el separador de directoris en Java, el mateix que Unix. Java és capaç de traduir una ruta amb aquest separador als altres sistemes operatius de forma automàtica.



Crear un objecte File **no crea cap fitxer a l'ordinador**. Es pot crear una instància per qualsevol nom de fitxer independentment de si existeix o no.

El mètode **exists()** de la classe File permet veure si el fitxer existeix o no.

File(pathname: String)

Crea un objecte File per la ruta especificada, *pathname* pot ser un fitxer o un directori.

File(parent: String, child: String)

Crea un objecte File per child dins del directori parent. child pot ser un fitxer o un directori.

File(parent: File, child: String)

Crea un objecte File per child dins del directori parent. parent és un objecte File.

63.5. Comprovar si el fitxer existeix

El mètode **exists()** de la classe **File** permet determinar si la ruta encapsulada a l'objecte **File** existeix o no.

exists(): boolean

Retorna **true** si el fitxer o directori representat per l'objecte File existeix.

```
File testFile = new File("dummy.txt");

boolean fileExists = testFile.exists();
if (fileExists) {
    System.out.println("testFile.txt existeix.");
} else {
    System.out.println("textFile.txt no existeix.");
}
```

63.6. Comprovar si el fitxer és un directori

Un objecte de tipus **File** pot ser tant un directori, com un fitxer. Per determinar què és es poden utilitzar els mètodes **isDirectory()** i **isFile()**.

isDirectory

Retorna true si el fitxer indicat per la ruta és un directori.

isFile()

Retorna true si el fitxer indicat per la ruta és un fitxer.

63.7. Ruta absoluta i ruta canònica

Una **ruta absoluta** identifica un fitxer dins un sistema de fitxers. El problema és que es poden construir moltes rutes absolutes que fan referència al mateix fitxer.

I això pot ser un problema, en particular si es vol determinar si dues rutes són la mateixa.

Per exemple, les següents rutes fan referència al mateix fitxer:

```
C:/test/testFile.txt  
C:/test/temporals/../testFile.txt  
C:/test/temporals/../temporals/../testFile.txt
```

Una **ruta canònica** és la ruta absoluta més simple que fa referència a un fitxer.

Els mètodes **getAbsolutePath()** i **getCanonicalPath()** de la classe **File** retornen la ruta absoluta i la ruta canònica de la ruta encapsulada dins del objecte **File**.

getAbsolutePath(): String

Retorna la ruta sencera del fitxer o directori representat per l'objecte File.

getCanonicalPath(): String

Retorna el mateix que el mètode getAbsolutePath però elimina tota la informació redundant de la ruta.

Elimina els . i .., resol els **links simbòlics**, en Linux, i posa la **lletra de la unitat en majúscula**, en Windows.

Per exemple:

```
import java.io.File;  
import java.io.IOException;  
  
public class FileDemo {  
  
    public static void main(String[] args) throws IOException {  
        File f1 = new File("/test/../test/fitxer.txt");  
  
        System.out.println("Name: " + f1.getName());  
        System.out.println("Path: " + f1.getPath());  
        System.out.println("Absolute Path: " + f1.getAbsolutePath());  
        System.out.println("Canonical Path: " + f1.getCanonicalPath());  
        System.out.println("Parent: " + f1.getParent());  
        System.out.println(f1.exists() ? "exists" : "does not exist");  
        System.out.println(f1.canWrite() ? "is writeable" : "is not  
writeable");  
    }  
}
```

```

        System.out.println(f1.canRead() ? "is readable" : "is not
readable");
        System.out.println("is " + (f1.isDirectory() ? "" : "not" + " a
directory"));
        System.out.println(f1.isFile() ? "is normal file" : "might be a
named pipe");
        System.out.println(f1.isAbsolute() ? "is absolute" : "is not
absolute");
        System.out.println("File last modified: " + f1.lastModified());
        System.out.println("File size: " + f1.length() + " Bytes");
    }

}

```

Name: fitxer.txt
Path: \test\..\test\fitxer.txt
Absolute Path: C:\test\..\test\fitxer.txt
Canonical Path: C:\test\fitxer.txt
Parent: \test\..\test
does not exist
is not writeable
is not readable
is not a directory
might be a named pipe
is not absolute
File last modified: 0
File size: 0 Bytes



No és recomanable treballar amb rutes absolutes dins dels programes, les rutes absolutes són susceptibles dels canvis de sistema operatiu, dels canvis de directori de treball i dels canvis d'ubicació del programa.

Si es treballa amb rutes absolutes, aquestes haurien d'emmagatzemar fora del programa, en un fitxer de configuració, passades com a paràmetres d'inici, etc...

63.8. Crear, renombrar i eliminar fitxers

createNewFile(): boolean

El mètode **createNewFile()** crea un fitxer nou, buit, si el nom especificat a l'objecte File no existeix. El mètode llança una excepció de tipus **IOException** si es produeix un error d'entrada/sortida.

Per exemple:

```
File dummyFile = new File("dummy.txt");
boolean fileCreated = dummyFile.createNewFile();
```

delete(): boolean

El mètode **delete()** elimina el fitxer o directori especificat. Els directoris han d'estar buits per a poder-se eliminar. El mètode retorna **true** si el fitxer o directori s'ha eliminat correctament, en cas contrari retorna **false**.

Per exemple:

```
File dummyFile = new File("dummy.txt");
dummyFile.delete();
```

mkdir(): boolean

Crea el directori representat per l'objecte File. Retorna **true** si el directori es crea amb èxit.

mkdirs(): boolean

Els mètodes **mkdir()** i **mkdirs()** creen nous directoris. El mètode **mkdir()** crea un nou directori si el directori pare especificat a la ruta existeix. El mètode **mkdirs()** crea el directori si el directori pare no existeix.

Per exemple:

```
File newDir = new File("C:\\users\\home");
newDir.mkdir(); // Crea el directori home si existeix c:\\users
newDir.mkdirs(); // Crea el directori c:\\users\\home si c:\\users no
existeix
```

renameTo(dest: File): boolean

El mètode **renameTo()** permet canviar el nom d'un fitxer. Aquest mètode retorna **true** si el fitxer s'ha eliminat correctament, en cas contrari retorna **false**.

Per exemple:

```
File oldFile = new File("old_dummy.txt");
File newFile = new File("new_dummy.txt");
boolean fileRenamed = oldFile.renameTo(newFile);
if (fileRenamed) {
    System.out.println(oldFile + " renamed to " + newFile);
} else {
    System.out.println("Renaming " + oldFile + " to " + newFile + " failed.");
}
```

63.9. Treballar amb els atributs dels fitxers

La classe **File** conté alguns mètodes que permeten manipular els atributs dels fitxers de forma limitada.

setReadonly(): boolean

Marca el fitxer o el directori com a només lectura.

setReadable(boolean readable): boolean

Proporciona permisos de lectura pel propietari del fitxer. Retorna true si l'operació ha tingut èxit.

setWritable(boolean writable): boolean

Proporciona permisos d'escriptura pel propietari del fitxer. Retorna true si l'operació ha tingut èxit.

setExecutable(boolean executable): boolean

Proporciona permisos d'execució pel propietari del fitxer. Retorna true si l'operació ha tingut èxit.

Per exemple:

```
import java.io.File;
```

```
public class FileTest {  
  
    public static void main(String[] args) {  
  
        File f = null;  
        boolean bool = false;  
  
        // Creem un objecte File  
        f = new File("C:/test.txt");  
        if (f.exists()) {  
            // Establism permís de lectura  
            bool = f.setReadable(true);  
  
            System.out.println("setReadable() establert?: " + bool);  
  
            // Comprovem si el fitxer té permisos de lectura  
            bool = f.canRead();  
  
            System.out.print("El fitxer té permisos de lectura?: " +  
bool);  
        }  
    }  
}
```

63.10. Copiar un fitxer

La classe **File** no proporciona cap mètode que permeti copiar un fitxer.

Per copiar un fitxer cal crear-ne un de nou, llegir el contingut del fitxer original i escriure'l al nou fitxer.

Com alternativa és pot utilitzar el mètode **copy** de la classe **java.nio.Files** que proporciona varies sobrecàrregues.

63.11. Saber la mida d'un fitxer

El mètode **length()** retorna la mida d'un fitxer en bytes.

length(): long

Retorna la mida del fitxer en bytes.

63.12. Mostrar tots els directoris arrel

Es pot obtenir una llista de tots els directoris arrel d'un sistema de fitxers utilitzant el mètode **listRoots()**.

Els directoris arrel depenen del sistema operatiu, en Windows existeix un directori arrel per a cada unitat de disc (c:\, d:\, etc...) en Linux hi ha un únic directori arrel representant per (\)

```
import java.io.File;

public class RootList {

    public static void main(String[] args) {
        File[] roots = File.listRoots();
        System.out.println("Directoris arrel:");
        for (File f : roots) {
            System.out.println(f.getPath());
        }
    }
}
```

```
Directoris arrel:
C:\
D:\
G:\
Y:\
Z:\
```

63.13. Mostrar tots els fitxers i directoris d'un directori

La classe **File** proporciona mètodes per veure el contingut d'un directori.

listfiles(): File[]

Retorna una matriu de rutes indicant els fitxers del directori indicat pel path.

list(): String[]

Retorna una matriu de cadenes indicant els **noms** dels fitxers del directori indicat pel path.

```
import java.io.File;
```

```
public class FileTest {  
  
    public static void main(String[] args) {  
        File directoriArrel = new File(System.getProperty("user.dir"));  
        File[] fitxers = directoriArrel.listFiles();  
        for (int i = 0; i < fitxers.length; i++) {  
            if (fitxers[i].isDirectory()) {  
                System.out.print("[D] ");  
            } else if (fitxers[i].isFile()) {  
                System.out.print("[F] ");  
            } else {  
                System.out.print("[?] ");  
            }  
            System.out.println(fitxers[i].getName());  
        }  
    }  
}
```

```
[F] build.xml  
[F] f.txt  
[F] manifest.mf  
[D] nbproject  
[D] src  
[D] test
```


64

Lectura i escriptura de streams binaris d'accés seqüencial

En primer lloc anem a donar un cop d'ull a algunes de les classes que permeten treballar amb streams binaris de forma seqüencial.

64.1. Classe `java.io.InputStream`

La classe **InputStream** és la classe base de tots els streams d'entrada de la api **java.io**. Algunes de les seves subclasses són **FileInputStream**, **BufferedInputStream** i **PushbackInputStream** entre d'altres.

Un **InputStream** normalment està connectat a algun tipus d'origen de dades, un fitxer, una connexió de xarxa, un pipe, etc...

No es poden crear objectes de tipus **InputStream** directament cal fer-ho a partir d'alguna de les seves especialitzacions.

Per exemple, s'utilitza la classe **InputStream** per llegir dades basades en bytes:

```
File f = new File("c:\\\\data\\\\text.txt");
InputStream inputstream = new FileInputStream(f);

int data = inputstream.read();
while(data != -1) {
    //do something with data...
    doSomethingWithData(data);

    data = inputstream.read();
}
```

```
inputstream.close();
```

Aquest exemple crea una nova instància d'un **FileInputStream** que és una subclasse de **InputStream** i per tant té sentit **assignar una instància de FileInputStream a una variable de tipus InputStream**.



S'ha obviat la gestió de les excepcions a favor de la claredat de l'exemple.

64.1.1. Mètode read()

El mètode **read()** d'un **InputStream** retorna un **int** que conté el valor **d'un byte** del byte llegit.

Per exemple:

```
int data = inputstream.read();
```

Si convé, es pot fer un cast de l'enter retornat:

```
char aChar = (char) data;
```

Les subclasses de **InputStream** poden proporcionar mètodes **read()** alternatius. Per exemple, la classe **DataInputStream** permet llegir tipus primitius amb els corresponents mètodes **readBoolean()**, **readInt()**, **readDouble()**, etc...

64.1.2. Final del stream

Si el mètode **read()** retorna **-1** indica que s'ha arribat al final del stream de dades. És important recalcar que **el -1 retornat és un int**.

Quan s'ha arribat al final del stream cal tancar-lo cridant al mètode **close()**.

64.1.3. Mètode read(byte[])

La classe **InputStream** conté dos mètodes **read()** que permeten llegir dades d'un **InputStream** directament a una matriu de bytes. Aquests mètodes són:

- int read(byte[])
- int read(byte[], int offset, int length)

Llegir una matriu de bytes és molt més eficient que llegir un byte cada cop.

El mètode **read(byte[])** intentarà llegir tants bytes com espai tingui la matriu passada com a paràmetre. Aquest mètode retorna un **int** indicant la quantitat de bytes llegida. En cas que s'hagin llegit menys bytes que la capacitat de la matriu, la resta de la matriu seguirà contenint els mateixos elements que contingués prèviament.

El mètode **read(byte[], int offset, int length)** també llegeix bytes i els posa a la matriu passada com a paràmetre, però comença a la posició **offset** i llegeix com a molt **length** bytes a partir de la posició indicada. Igual que en el cas anterior el mètode retorna un **int** indicant la quantitat de bytes llegida.

Els dos mètodes **retornen -1** quan arriben al final del stream de dades.

Per exemple:

```
File f = new File("./text.txt");
InputStream inputstream = new FileInputStream(f);

byte[] data = new byte[1024];
int bytesRead = inputstream.read(data);

while(bytesRead != -1) {
    doSomethingWithData(data, bytesRead);

    bytesRead = inputstream.read(data);
}
inputstream.close();
```

64.1.4. Mètodes mark() i reset()

La classe **InputStream** té dos mètodes anomenats **mark()** i **reset()** que poden ser suportats o no per les seves subclasses.

Si una subclassa de **InputStream** suporta els mètodes anteriors cal que aquesta sobreescrigui el mètode **markSupported()** perquè retorni **true**.

El mètode **mark()** estableix una marca, internament, en el punt del stream fins a on s'ha llegit fins aquest moment. El codi que utilitza el **InputStream** pot seguir llegint dades amb normalitat. Si el codi que utilitza el **InputStream** vol anar fins el punt del stream on s'ha establert la marca pot fer-ho cridant el mètode **reset()**, en aquest moment el **InputStream** "rebobina" fins a situar-se exactament en el punt on s'ha establert la marca i **llegirà les dades a partir d'aquest punt** altre cop.

Els mètodes **mark()** i **reset()** s'utilitzen típicament al implementar parsers, per exemple.

64.1.5. Exemple 1

System.in és un **InputStream**, el següent exemple agafa bytes de l'entrada estàndard, els mostra per pantalla i para quan el byte entrat és 65 que correspon a la lletra **A** en ASCII.

```
import java.io.IOException;
import java.io.InputStream;

public class StreamsDemo {

    public static void main(String[] args) throws IOException {
        InputStream is = System.in;

        int nextByte;
        do {
            nextByte = is.read();
            System.out.print(nextByte + "(" + (char) nextByte + ")");
        } while (nextByte != 65); // ASCII 65 = A
    }
}
```

64.1.6. Exemple 2

En aquest exemple accedim a una cadena byte a byte amb l'ajuda d'un **InputStream**.

```
import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.InputStream;
```

```

public class StreamsDemo {

    public static void main(String[] args) throws IOException {
        InputStream is = new
ByteArrayInputStream("ABCDEFGHIJKLMNOPQRSTUVWXYZ".getBytes());

        int nextByte;
        do {
            nextByte = is.read();
            System.out.print(nextByte + "(" + (char) nextByte + ") ");
        } while (nextByte != -1);
    }
}

```

64.2. Exemple 3

Mostra el contingut d'un fitxer de text en hexadecimal.

```

import java.io.ByteArrayInputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;
import java.util.logging.Level;
import java.util.logging.Logger;

public class StreamsDemo {

    public static void main(String[] args) {

        try (InputStream is = new FileInputStream("manifest.mf")) {
            int nextByte;
            String mask = "00000000";
            do {
                nextByte = is.read();
                System.out.printf("%2h ", nextByte);

            } while (nextByte != -1);
        } catch (FileNotFoundException ex) {
            System.out.println(ex.toString());
        } catch (IOException ex) {
            System.out.println(ex.toString());
        }
    }
}

```

```

    }
}
```

64.3. Classe Java.io.OutputStream

La classe **OutputStream** és la classe base de tots els streams d'escriptura de java.io. Algunes de les seves subclasses són **BufferedOutputStream** i **FileOutputStream** entre d'altres.

No es poden crear objectes de tipus OutputStream directament cal fer-ho a partir d'alguna de les seves especialitzacions.

64.3.1. Mètode write(int)

El mètode **write(int)** s'utilitza per escriure un únic **byte** al **OutputStream**. Aquest mètode agafa un **int** que **conté el valor d'un byte** i l'escriu al stream de sortida. **Només s'escriu el primer byte del valor int la resta de bytes s'ignoren.**

Les subclasses de **OutputStream** poden tenir mètodes **write** alternatius. Per exemple, la classe **DataOutputStream** permet escriure valors primitius directament a un stream a través dels mètodes **writeBoolean()**, **writeDouble()**, etc...

Per exemple:

```

OutputStream output = new FileOutputStream("c:\\\\data\\\\output-text.txt");

while(hasMoreData()) {
    int data = getMoreData();
    output.write(data);
}
output.close();
```



S'ha obviat la gestió de les excepcions a favor de la claredat de l'exemple.

La classe **OutputStream** té un mètode **write(byte[] bytes)** i un mètode **write(byte[] bytes, int offset, int length)** que permeten escriure una matriu, o part d'una matriu, de bytes al **OutputStream**.

64.3.2. Mètode flush()

El mètode **flush()** **força l'escriptura de totes les dades escriptes al OutputStream a la destinació subjacent**. Per exemple, si el **OutputStream** és un **FileOutputStream** es possible que els bytes escrits al **OutputStream** encara no s'hagin gravat completament al disc, a lo millor s'han emmagatzemat en un buffer a la memòria de l'ordinador, cridar al mètode **flush()** es garanteix que totes les dades emmagatzemades en buffers intermitjós s'escriuran al disc (o a la destinació adient).

64.3.3. Mètode close()

Quan s'ha acabat d'escriure dades en un **OutputStream** cal tancar-lo i alliberar-ne els recursos i els possibles bloquejos del destí de dades subjacent. Per fer-ho cal cridar al mètode **close()**.

Com que els mètodes **write()** del stream poden llançar excepcions de tipus **IOException**, cal tancar el OutputStream dins d'un bloc **finally**.

Per exemple:

```
OutputStream output = null;

try{
    output = new FileOutputStream("c:\\\\data\\\\text.txt");

    while(hasMoreData()) {
        int data = getMoreData();
        output.write(data);
    }
} finally {
    if(output != null) {
        output.close();
    }
}
```

64.4. Classe java.io.FileInputStream i FileOutputStream

Per llegir dades binàries d'un fitxer en un disc cal crear un objecte **FileInputStream**.

```
FileInputStream inputStream = new FileInputStream("input.bin");
```

De forma similar, si el que volem es escriure dades en un fitxer binari en un disc, utilitzem la classe **FileOutputStream**

```
FileOutputStream outputStream = new FileOutputStream("output.bin");
```



Per defecte **FileInputStream** escriu els bytes al principi del fitxer, per afegir bytes **al final** del fitxer cal crear l'objecte de la següent manera:

```
FileOutputStream outputStream = new  
FileOutputStream("output.bin", true);
```

La classe **FileInputStream** té un mètode, **read**, que permet llegir un byte cada cop, la classe **FileInputStream** permet que aquest byte es llegeixi des d'un fitxer de disc.

El mètode **read** retorna un **int**, no un byte, d'aquesta manera pot indicar que s'ha arribat al final del fitxer retornant un **-1**, en cas contrari retorna un valor entre 0 i 255.

```
FileInputStream in = new FileInputStream("input.bin");  
int next = in.read();  
if (next != -1) {  
    // Processem les dades  
}
```

De forma similar, la classe **FileOutputStream** disposa del mètode **write** que permet escriure un sol byte al fitxer. Retorna un **int** però només n'utilitza els 8 bits menys significatius.

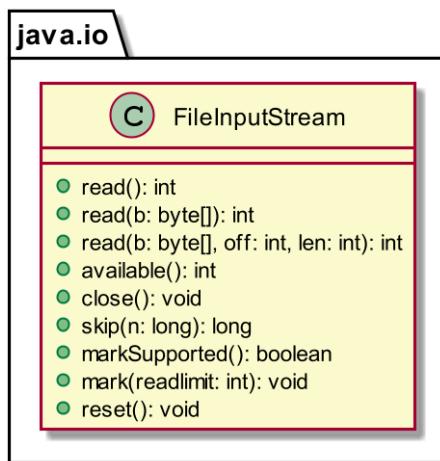
```
FileOutputStream out = new FileOutputStream("output.bin");  
int valor = 45; // valor entre 0 i 255  
out.write(valor);
```

En acabar de processar les dades caldrà tancar els fluxos de dades cridant al mètode **close**.

```
in.close();
out.close();
```

Aquests són els únics mètodes que proporcionen les classes **FileInputStream** i **FileOutputStream**, si es volen llegir números, cadenes o objectes caldrà utilitzar alguna de les classes amb capacitat d'agrupar bytes individuals en estructures més complexes.

64.5. Mètodes de la classe InputStream



`read(): int`

Llegeix el següent byte del flux d'entrada. El valor retornat és un int entre 0 i 255. Si s'ha arribat al final del flux es retorna -1.

`read(b: byte[]): int`

Llegeix fins a `b.length` del flux d'entrada i retorna la quantitat de bytes llegits. Retorna -1 al final del flux.

`read(b: byte[], off: int, len: int): int`

Llegeix bytes del flux d'entrada i els emmagatzema a `b[off]`, `b[off+1]`, ..., `b[off +len-1]`. Retorna la quantitat de bytes llegits. Retorna -1 al final del flux.

`available(): int`

Retorna la quantitat de bytes estimats que es poden llegir al flux d'entrada.

close(): void

Tanca el flux d'entrada i allibera els recursos associats.

skip(n: long): long

Salta i descarta n bytes de dades. Es retorna la quantitat de bytes descartats.

markSupported(): boolean

Determina si el flux suporta els mètodes **mark** i **reset**.

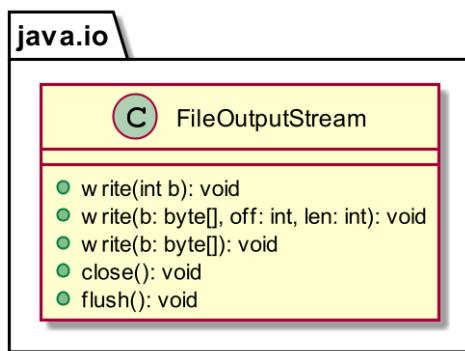
mark(readlimit: int): void

Marca la posició actual en el flux d'entrada.

reset(): void

Reposiciona l'apuntador del flux a la última posició definida pel mètode **mark**.

64.6. Mètodes de la classe OutputStream

**write(int b): void**

Escriu el byte especificat al flux de sortida. El paràmetre b és de tipus int.

write(b: byte[], off: int, len: int): void

Escriu `b[off]`, `b[off+1]`, ..., `b[off+len-1]` al flux de sortida.

write(b: byte[]): void

Escriu tots els bytes de la matriu al flux de sortida.

close(): void

Tanca el flux de dades i n'allibera tots els recursos associats.

flush(): void

Buida el flux de sortida i força l'escriptura de tots els bytes del buffer.

64.6.1. Exemple Xifrat del cèsar

El següent programa encripta un fitxer agafant cadascun dels bytes i sumant-els-hi n tenint en compte que si el valor resultant sobrepassa 255 es dona la volta començant altre cop per 0.

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.util.Scanner;

/**
 * Aquest programa encripta un fitxer utilitzant el xifrat del cèsar.
 */
public class CaesarEncryptor {

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        try {
            System.out.print("Fitxer d'entrada: ");
            String inFile = in.nextLine();
            System.out.print("Fitxer de sortida: ");
            String outFile = in.nextLine();
            System.out.print("Clau d'encriptació: ");
            int clau = in.nextInt();

            FileInputStream inStream = new FileInputStream(inFile);
            FileOutputStream outStream = new FileOutputStream(outFile);

            encriptarFlux(clau, inStream, outStream);
            inStream.close();
            outStream.close();
        } catch (IOException ex) {
            System.out.println("Error processant el fitxer: " + ex);
        }
    }

    /**
     * Encripta el contingut d'un flux.
    
```

```
* @param clau clau d'encriptació
* @param in flux d'entrada
* @param out flux de sortida
*/
public static void encriptarFlux(int clau, FileInputStream in,
FileOutputStream out) throws IOException {
    boolean done = false;
    while (!done) {
        int next = in.read();
        if (next == -1) {
            done = true;
        } else {
            int encrypted = encriptar(next, clau);
            out.write(encrypted);
        }
    }
}

/**
 * Encripta un valor.
 *
 * @param b the value to encriptar (between 0 and 255)
 * @return the encrypted value
 */
public static int encriptar(int b, int clau) {
    return (b + clau) % 256;
}
```

65

Lectura i escriptura de fitxers de text a baix nivell

Les classes presentades en aquest apartat representen les classes més bàsiques d'accés a fitxers de text.

Es caracteritzen perquè treballen amb un caràcter cada vegada, tant a nivell de lectura com a nivell d'escriptura.

65.1. Tractament de possibles errors d'accés al fitxer

Hi ha un problema addicional amb el que hem de tractar. Si el fitxer al que volem accedir no existeix es produirà una excepció de tipus **FileNotFoundException** i si intentem llegir/escriure dades del fitxer es pot produir una excepció de tipus **IOException**.

El problema és que el compilador insisteix en que especifiquem què és el que ha de fer el programa quan es produueixen aquestes excepcions, si no li especificuem el programa no es deixa compilar.

Per sortir del pas indicarem al compilador que si no es troba el fitxer indicat es dispari l'excepció en forma d'error d'execució, s'interrompi l'execució i es mostri la informació de l'excepció per la sortida estàndard i que actuï de la mateixa manera si es produex una excepció quan s'intenta la lectura d'un valor.

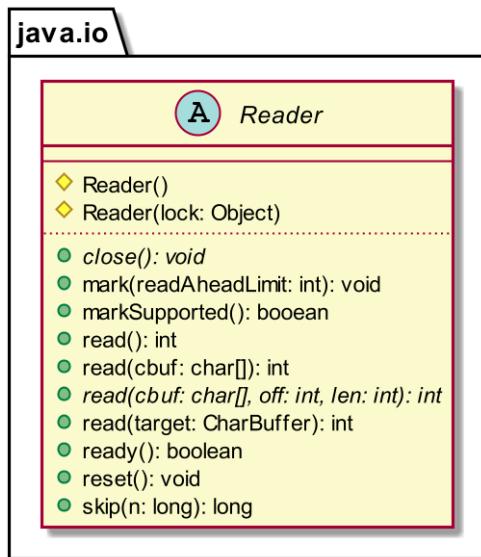
Per fer-ho cal declarar la funció *main* de la següent manera:

```
public static void main(String[] args) throws FileNotFoundException,  
IOException {
```

```
}
```

Cal notar que aquesta no és una manera bona de procedir, el que voldrem aconseguir serà gestionar l'excepció i gestionar la condició de l'error, no volem la interrupció definitiva del codi.

65.2. Classe java.io.Reader



La classe **Reader** és la classe base per a totes les classes **Reader** de **java.io**. Un **Reader** està dissenyat per a llegir text i representa la part comú de tota la jerarquia de "readers", és a dir, **tots els readers disposen dels mètodes mostrats al diagrama anterior**.

65.2.1. Caràcters i Unicode

A dia d'avui, moltes aplicacions utilitzen UTF (UTF-8 o UTF-16) per emmagatzemar dades en format text.

Per exemple, en UTF-8 els caràcters ocupen **un o dos bytes**, en UTF-16 els caràcters ocupen **2 bytes**.

Per tant, l'hora de llegir text és molt complicat fer-ho byte a byte.

Per solucionar-ho tenim la classe **Reader**. La classe **Reader** és capaç de descodificar bytes en caràcters, per fer-ho, **cal informar-li de la codificació de caràcters a utilitzar** just en el moment de la seva creació.

65.2.2. Llegir caràcters amb un Reader

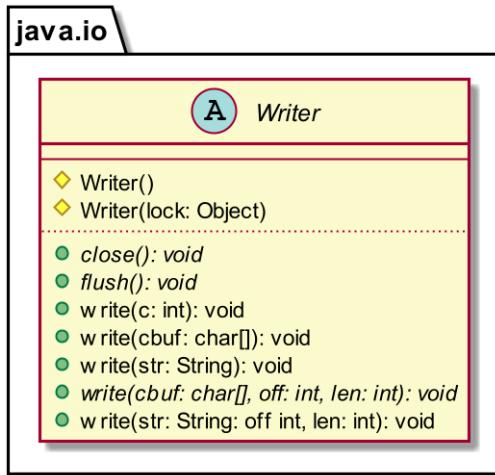
El mètode **read()** d'un **Reader** retorna un **int** que conté el següent el valor del següent **char** llegit.

Si el mètode **read()** retorna -1 indica que no hi ha més dades per llegir i per tant es pot tancar amb el mètode **close()**.

Els **Reader** no es poden crear directament, cal crear alguna de les seves subclasses en funció de les funcionalitats desitjades.

Un objecte **Reader** sol estar vinculat a algun **origen de dades**, aquest origen de dades pot ser un fitxer, una cadena, una matriu de **char**, un socket de xarxa, etc...

65.3. Classe java.io.Writer



La classe **Writer** és la classe base per a totes les classes **Writer** de **java.io**. Un **Writer** és similar a un **OutputStream** amb la diferència que està basada en caràcters enllloc d'estar basada en bytes. És a dir, un **Writer** està dissenyat per a escriure text mentre que un **OutputStream** està dissenyat per a escriure bytes.

65.3.1. Caràcters i Unicode

A dia d'avui, moltes aplicacions utilitzen UTF (UTF-8 o UTF-16) per emmagatzemar dades en format text.

Per exemple, en UTF-8 els caràcters ocupen **un o dos bytes**, en UTF-16 els caràcters ocupen **2 bytes**.

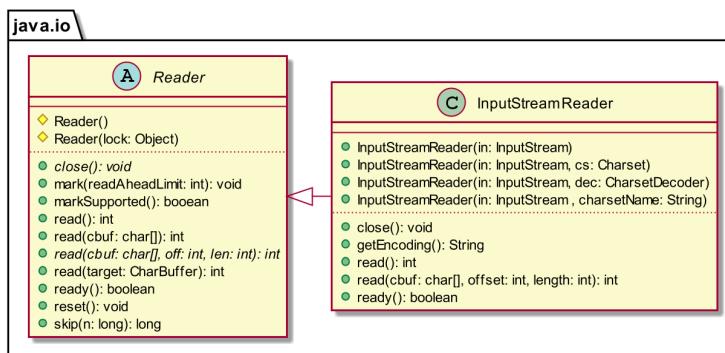
Per tant, l'hora d'escriure text és complicat fer-ho byte a byte.

Per solucionar-ho tenim la classe **Writer**. Les subclasses de la classe **Writer** permeten tractar les codificacions més habituals internament.

Els **Writer** no es poden crear directament, cal crear alguna de les seves subclasses en funció de les funcionalitats desitjades.

Un objecte **Writer** sol estar vinculat a algun **destí de dades**, aquest origen de dades pot ser un fitxer, una matriu de **char**, un socket de xarxa, etc...

65.4. Classe java.io.InputStreamReader



Com a especialització de la classe **Reader** hi ha la classe **InputStreamReader**, la característica principal d'aquest "reader" es que es construeix a partir d'una de les classes **Stream**, que recordem, servien per tractar els fluxos de dades byte a byte.

Per tant, l'objectiu d'aquesta classe és **adaptar un objecte capaç de llegir un flux byte a byte en un objecte que es capaç de llegir el mateix flux de dades però caràcter a caràcter enlloc de byte a byte**.

Per exemple:

```

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStreamReader;

public class FileTest {

    public static void main(String[] args) throws FileNotFoundException,
    IOException {
        File f = new File("c:\\\\data\\\\input.txt"); ①
        FileInputStream inputStream = new FileInputStream(f); ②
        InputStreamReader inputStreamReader = new
        InputStreamReader(inputStream); ③

        int data = inputStreamReader.read();
        while (data != -1) {
            char caracterLlegit = (char) data; ④
            data = inputStreamReader.read();
        }

        inputStreamReader.close();
    }
}

```

- ① Determinem el fitxer al que volem accedir
- ② Accedim al fitxer seqüencialment i byte a byte amb un **FileInputStream**.
- ③ Creem un nou **InputStreamReader** a partir del **FileInputStream** anterior.
- ④ Ara podem accedir al fitxer caràcter a caràcter.



La classe **InputStreamReader** s'acostuma a utilitzar per a fitxers (o connexions de xarxa) on els bytes representen text. Per exemple, un fitxer de text amb els caràcters codificats en UTF-8. En aquest cas es pot utilitzar un **InputStreamReader** per encapsular un **FileInputStream** i així poder llegir el fitxer com a text.

65.4.1. Mètode read()

El mètode **read()** d'un **InputStreamReader** retorna un **int** que conté el valor del **char** llegit.

Fixeu-vos que **read()** retorna un **int** enlloc d'un **char** que és el què esperàvem. Això és així per permetre detectar el final de fitxer mentre es realitza una lectura seqüencial.

65.4.2. Final de fitxer

Recordeu que un **char** sempre és un **número positiu**.

Si el mètode **read()** retorna **-1**, cal un **int** per emmagatzemar el -1, vol dir que s'ha arribat al final del fitxer, en aquest cas es pot tancar el **InputStreamReader** cridant al seu mètode **close()**.

Per tant, per llegir un fitxer a través d'un **InputStreamReader** cal:

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStreamReader;

public class FileTest {

    public static void main(String[] args) throws FileNotFoundException,
    IOException {
        File f = new File("./data/input.txt");
        FileInputStream inputStream = new FileInputStream(f);
        InputStreamReader inputStreamReader = new
        InputStreamReader(inputStream);

        int data = inputStreamReader.read(); ①
        while (data != -1) { ②
            char caracterLlegit = (char) data; ③
            data = inputStreamReader.read();
        }

        inputStreamReader.close();
    }
}
```

```

    }
}

```

- ① Llegir el següent caràcter com a **int**.
- ② Comprovar si la lectura és -1 i per tant hem arribat a final de fitxer.
- ③ Si no estem a final de fitxer convertim el **int** llegit en el pas anterior en un **char** i el tractem convingui.

Selecció de la codificació de caràcters

A l'hora de crear un nou **InputStreamReader** es pot indicar quina serà la codificació de caràcters emprada, per exemple:

```

InputStream inputStream = new FileInputStream("c:\\\\data\\\\input.txt");
Reader inputStreamReader = new InputStreamReader(inputStream, "UTF-8");

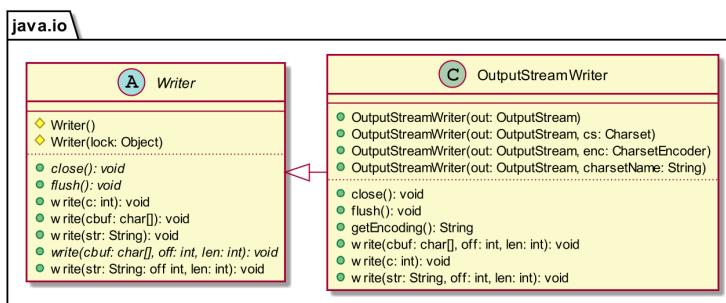
```

65.4.3. Tancar un InputStreamReader

Cal alliberar els recursos i bloqueigos associats al fitxer en acabar d'utilitzar-lo, per fer-ho, cal cridar al mètode **close()**.

Tancar un **InputStreamReader** tanca automàticament el **InputStream** encapsulat.

65.5. Classe java.io.OutputStreamWriter



La classe **OutputStreamWriter** està pensada per encapsular un **OutputStream**, d'aquesta manera es pot convertir un stream d'escriptura basat en bytes a un **Writer** basat en caràcters.

Per tant, l'objectiu d'aquesta classe és **adaptar un objecte capaç d'escriure a un flux byte a byte en un objecte que es capaç d'escriure sobre el mateix flux de dades però caràcter a caràcter**.

Per exemple:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStreamWriter;

public class FileTest {

    public static void main(String[] args) throws FileNotFoundException,
IOException {
        File f = new File("data/output.txt"); ①
        FileOutputStream outputStream = new FileOutputStream(f); ②
        OutputStreamWriter outputStreamWriter = new
OutputStreamWriter(outputStream); ③
        outputStreamWriter.write("Hello World!!"); ④
        outputStreamWriter.close(); ⑤
    }
}
```

- ① Determinem el fitxer al que volem accedir.
- ② Accedim al fitxer seqüencialment i byte a byte amb un **FileOutputStream**.
- ③ Creem un nou **OutputStreamWriter** a partir del **FileOutputStream** anterior.
- ④ Escrivim al fitxer caràcter a caràcter, en aquest cas passem una cadena que serà guardada caràcter a caràcter.
- ⑤ Tanquem el "writer".

65.5.1. Determinar la codificació de caràcters

La classe **OutputStreamWriter** permet triar la codificació de caràcters que s'utilitzarà per escriure al **OutputStream** encapsulat. Per exemple:

```
OutputStream outputStream = new FileOutputStream("c:\\\\data\\\noutput.txt");
```

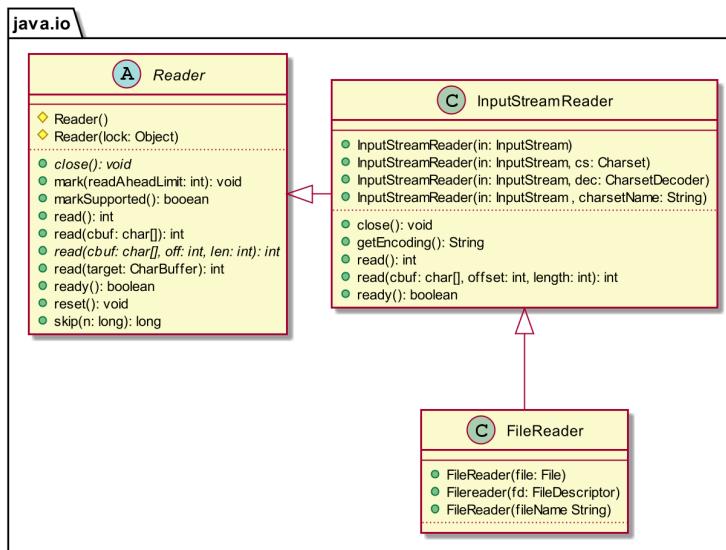
```
Writer outputStreamWriter = new OutputStreamWriter(outputStream, "ISO-  
Latin1");
```

65.5.2. Tancar un OutputStreamWriter

Cal alliberar els recursos i bloqueigs associats al fitxer en acabar d'utilitzar-lo, per fer-ho, cal cridar al mètode **close()**.

Tancar un **OutputStreamReader** tanca automàticament el **OutputStream** encapsulat.

65.6. Classe java.io.FileReader



La classe **FileReader** permet llegir el contingut d'un fitxer caràcter a caràcter, és una especialització de la classe **InputStreamReader** amb la única diferència que es construeix directament a partir d'un objecte **File** enlloc d'un objecte **InputStream**.

Per exemple, enlloc de fer:

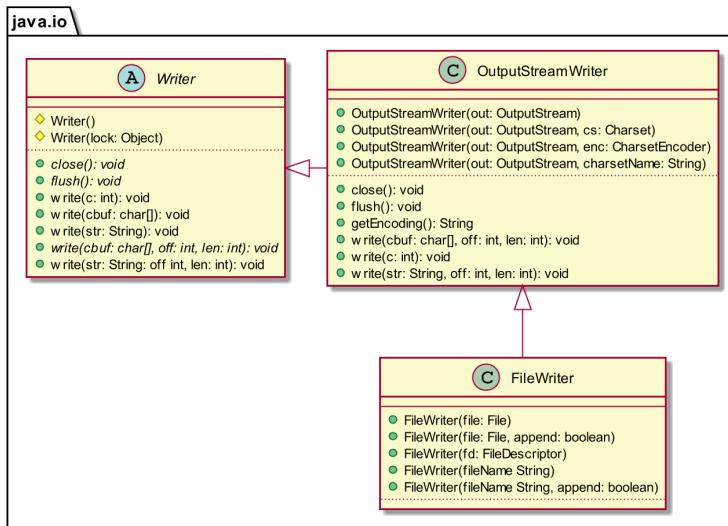
```
File f = new File("./data/input.txt");
FileInputStream fis = new FileInputStream(f);
InputStreamReader isr = new InputStreamReader(fis);
```

Fariem directament:

```
File f = new File("./data\\input.txt");
FileReader fr = new FileReader(f);
```

Pel demés el funcionament és exactament el mateix que el de la classe **InputStreamReader**.

65.7. Classe java.io.FileWriter



La classe **FileWriter** permet escriure caràcters a un fitxer. És una especialització de la classe **OutputStreamWriter** amb la única diferència que es construeix directament a partir d'un objecte **File** enlloc d'un objecte **OutputStream**.

Per exemple, enlloc de fer:

```
File f = new File("./data\\input.txt");
FileOutputStream fos = new FileOutputStream(f);
OutputStreamWriter osw = new OutputStreamWriter(fos);
```

Fariem directament:

```
File f = new File("./data\\input.txt");
```

```
FileWriter fr = new FileWriter(f);
```

Pel demés el funcionament és exactament el mateix que el de la classe **OutputStreamWriter**.

65.7.1. Sobreescriure el fitxer o afegir dades al final

Quan es crea un **FileWriter** apuntant a un fitxer que ja conté dades cal decidir si es vol sobreescriure el fitxer o bé afegir les noves dades al final.

Per fer-ho cal passar un paràmetre addicional a l'hora de crear l'objecte **FileWriter**, aquest paràmetre és un **boolean** que indica si el fitxer s'obre per a sobreescriptura, **false**, o bé per afegir dades, **true**.

Per exemple:

```
FileWriter fileWriter = new FileWriter("c:\\\\data\\\\output.txt", true);
//afegeix dades al fitxer

FileWriter fileWriter = new FileWriter("c:\\\\data\\\\output.txt",
false); // sobreescriu el fitxer
```



Si no s'especifica el paràmetre booleà el fitxer s'obre sobreescrivint les dades originals.

```
FileWriter fileWriter = new FileWriter("c:\\\\data\\\\output.txt");
```

65.8. La classe java.io.BufferedReader

La classe **BufferedReader** llegeix text des d'un stream de caràcters, de forma similar a un **Reader**. La seva característica diferencial és que gestiona un buffer en memòria fent més eficient la lectura de caràcters en quan aquesta lectura és poc eficient en termes de rendiment quan es realitza directament del **Reader****.

La mida del buffer es pot especificar o utilitzar la mida per defecte que sol ser suficient en la majoria de les situacions.

Un exemple d'utilització podria ser el següent:

```
import java.io.*;
public class LlegirFitxer {
    public static void main(String[] args) throws Exception {

        File file = new File("test.txt");
        BufferedReader br = new BufferedReader(new FileReader(file)); ①

        String st;
        while ((st = br.readLine()) != null) {
            System.out.println(st);
        }
    }
}
```

- ① Encapsulem un **FileReader** dins d'un **BufferedReader**.

66

Lectura i escriptura de fitxers de text a alt nivell

Treballar amb fitxers a baix nivell pot ser útil a vegades però normalment voldrem classes per permetir llegir i escriure tipus de dades primitius directament, enlloc de només un caràcter cada vegada.

Per fer-ho, utilitzarem la classe **Scanner** per la lectura de fitxers de text i la classe **PrintWriter** per la seva escriptura.

66.1. Classe Scanner

Recordem que l'objecte Scanner encapsula un flux de dades d'entrada proporcionat just a la creació de l'objecte. Si es vol llegir dades des de la consola caldrà encapsular el flux **System.in**.

```
Scanner in = new Scanner(System.in);
```

De forma similar es pot llegir el contingut d'un fitxer de text amb un objecte Scanner, només fa falta obtenir un flux de dades de lectura al fitxer, això ho proporciona la classe **File**. Per tant, caldrà:

```
File inputFile = new File("input.txt");
Scanner in = new Scanner(inputFile);
```

Per detectar el final de fitxer es pot fer amb l'ajuda dels mètodes **hasNext()** i un bucle **while**.

Per exemple, per llegir dades d'un fitxer de text amb números es podria fer:

```
File inputFile = new File("input.txt");
Scanner in = new Scanner(inputFile);

while (in.hasNextDouble()) {
    double valor = in.nextDouble();
    // Processar el valor
}
```

Finalment caldria alliberar els recursos dedicats a mantenir el fitxer de text obert, per fer-ho és important cridar el mètode **close()** de l'objecte Scanner.

```
in.close();
```

Anem a veure un exemple sencer d'utilització de la classe Scanner per a la lectura d'un fitxer de text.

66.2. Exemple lectura d'un fitxer de text mitjançant la classe Scanner

Volem llegir fitxers amb la següent estructura:

```
Pere López Antuno 8 María Ferrer Ortuño 5 Eva Gusi Pons
```



Hi ha un problema addicional amb el que hem de tractar. Si el fitxer al que volem accedir no existeix es produirà una excepció de tipus **FileNotFoundException** quan es construeixi l'objecte Scanner.

El problema és que el compilador insisteix en que especificuem què és el que ha de fer el programa quan es produeix aquesta excepció, si no li especificuem el programa no es deixa compilar.

Per sortir del pas indicarem al compilador que si no es troba el fitxer indicat es dispari l'excepció en forma d'error d'execució, s'interrompi l'execució i es mostri la informació de l'excepció per la sortida estàndard.

Exemple lectura d'un fitxer de
text mitjançant la classe Scanner

Per fer-ho cal declarar la funció *main* de la següent manera:

```
public static void main(String[] args) throws  
FileNotFoundException {  
}
```

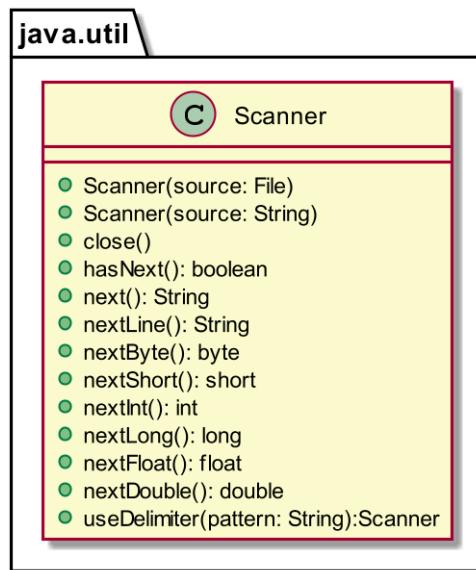
Cal notar que aquesta no és una manera bona de procedir, el que voldrem aconseguir serà gestionar l'excepció i gestionar la condició de l'error, no volem la interrupció definitiva del codi.

```
import java.io.FileNotFoundException;  
import java.io.File;  
import java.util.Scanner;  
  
public class ReadData {  
  
    public static void main(String[] args) throws FileNotFoundException  
{  
  
        file = new File("notes.txt");  
        Scanner in = new Scanner(file);  
  
        while (in.hasNext()) {  
            String nom = in.next();  
            String primerCognom = in.next();  
            String segonCognom = in.next();  
            int nota = in.nextInt();  
            System.out.println(nom + " " + primerCognom + " " +  
segonCognom + " " + nota);  
        }  
  
        // No oblidem tancar el fitxer!!  
        in.close();  
    }  
}
```



Què passaria si el nom d'un alumne fos Josep Maria?

66.3. Mètodes de la classe Scanner



Scanner(source: File)

Crea un objecte Scanner a partir del fitxer especificat.

Scanner(source: String)

Crea un objecte Scanner a partir de la cadena especificada.

close()

Tanca l'objecte Scanner.

hasNext(): boolean

Retorna true si el Scanner té més dades per ser llegides.

next(): String

Retorna el següent valor com a String.

nextLine(): String

Retorna un String fins a un separador de línia.

nextByte(): byte

Retorna el següent valor com a byte.

nextShort(): short

Retorna el següent valor com a short.

nextInt(): int

Retorna el següent valor com a int.

nextLong(): long

Retorna el següent valor com a long.

nextFloat(): float

Retorna el següent valor com a float.

nextDouble(): double

Retorna el següent valor com a double.

useDelimiter(pattern: String):Scanner

Estableix quin és el patró separador de valors i retorna el propi Scanner.

66.4. Classe java.io.PrintWriter

La classe **PrintWriter** permet escriure dades **amb format** al **Writer** subjacent. Per exemple, permet escriure dades **int**, **long** i d'altres tipus primitius com a text enlloc d'escriure directament el seu valor en bytes.

És una classe molt útil per generar informes o documents similars on hi ha la necessitat de barrejar text i números. La classe **PrintWriter** té els mateixos mètodes que la classe **PrintStream** (System.out és un PrintStream) amb excepció dels que permeten escriure bytes directament.

66.5. Utilització d'un objecte PrintWriter

En primer lloc crearem un objecte **PrintWriter** a partir d'un objecte **File**.

```
File outputFile = new File("output.txt");
PrintWriter out = new PrintWriter(outputFile);
```

A continuació podem afegir dades al fitxer utilitzant els mètodes print proporcionats per PrintWriter.

```
out.println("Això és una prova");
```

```
out.printf("Total: %8.2f\n", total);
```

I finalment caldrà tancar el flux d'escriptura per alliberar els recursos associats i per garantir que tota la informació passada a l'objecte es traspassa efectivament al fitxer..

```
out.close();
```

Exemple d'utilització:

```
File file = new File("d:\\data\\report.txt");
FileWriter writer = new FileWriter(file);
PrintWriter printWriter = new PrintWriter(writer);

printWriter.print(true);
printWriter.print((int) 123);
printWriter.print((float) 123.456);

printWriter.printf(Locale.UK, "Text + data: %1$", 123);

printWriter.close();
```

66.6. Mètodes de la classe PrintWriter

java.io

C PrintWriter

- `PrintWriter(File)`
- `PrintWriter(File, Charset)`
- `PrintWriter(OutputStream)`
- `PrintWriter(OutputStream, AutoFlush)`
- `PrintWriter(String)`
- `PrintWriter(String, Charset)`
- `PrintWriter(Writer)`
- `PrintWriter(Writer, AutoFlush)`

- `PrintWriter(String)`
- `print(s: String): void`
- `PrintWriter(File)`
- `print(c: char): void`
- `print(cArray: char[]): void`
- `print(i: int): void`
- `print(l: long): void`
- `print(f: float): void`
- `print(d: double): void`
- `print(b: boolean): void`

També conté les versions sobrecarregades dels mètodes `println`.
També conté les versions sobrecarregades dels mètodes `printf`.

PrintWriter(filename: String)

Crea un objecte PrintWriter pel nom de fitxer especificat.

print(s: String): void

Escreu una cadena al fitxer.

PrintWriter(file: File)

Crea un objecte PrintWriter pel objecte file especificat.

print(c: char): void

Escreu un char al fitxer.

print(cArray: char[]): void

Escreu una matriu de char al fitxer.

print(i: int): void

Escreu un int al fitxer.

print(l: long): void

Escriu un long al fitxer.

print(f: float): void

Escriu un float al fitxer.

print(d: double): void

Escriu un double al fitxer.

print(b: boolean): void

Escriu un boolean al fitxer.

També conté les versions sobrecarregades dels mètodes println. També conté les versions sobrecarregades dels mètodes printf

El mètode println i printf funciona igual que el mètode de mateix nom de System.out.

66.7. Exemple d'utilització de PrintWriter

El següent exemple obra un fitxer de text i en fa una còpia modificant totes les lletres a majúscula.

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.Scanner;

public class Exercici {

    public static void main(String[] args) throws FileNotFoundException
    {

        Scanner console = new Scanner(System.in);
        System.out.print("Input file: ");
        String inputFileName = console.next();
        System.out.print("Output file: ");
        String outputFileName = console.next();

        File inputFile = new File(inputFileName);
        Scanner in = new Scanner(inputFile);
        PrintWriter out = new PrintWriter(outputFileName);
```

```
    while (in.hasNextLine()) {  
        String linia = in.nextLine();  
        out.println(linia.toUpperCase());  
    }  
  
    in.close();  
    out.close();  
}  
}
```

66.8. Llegir un fitxer de text per paraules

L'objectiu d'aquest apartat és fer un programa que donada una frase retorna les paraules.

Considerem el bucle següent:

```
while (in.hasNext()) {  
    String input = in.next();  
    System.out.println(input);  
}
```

Si l'usuari introduceix:

Pablito clavó un clavito, ¿qué clavito clavó Pablito?, el clavito que
Pablito clavó, era el clavito de Pablito.

la sortida serà:

```
Pablito  
clavó  
un  
clavito,  
¿qué  
clavito  
clavó  
Pablito?,  
el  
clavito  
que  
Pablito
```

```
clavó,  
era  
el  
clavito  
de  
Pablito.
```

El problema és que apareixen els signes de puntuació com si formessin part de les paraules que es volen separar.

A vegades es volen llegir les paraules i descartar qualsevol cosa que no sigui una lletra, per fer-ho es pot utilitzar el mètode **useDelimiter** de la classe Scanner i una **expressió regular**.

Per exemple:

```
in.useDelimiter("[^A-Za-z]+");  
while (in.hasNext()) {  
    String input = in.next();  
    System.out.println(input);  
}
```

Expressions regulars

Les expressions regulars descriuen patrons de lletres.

Per exemple, els números enters positius tenen un patró simple, poden contenir un o més dígits. L'expressió regular que descriu els números és [0-9]+.

El conjunt [0-9] denota qualsevol dígit entre 0 i 9 i el símbol + vol dir "**un o més**".

Les comandes de cerca dels editors de text solen entendre les expressions regulars, de fet, moltes altres utilitats fan ús d'aquestes expressions per localitzar patrons de text.

Per exemple el programa **grep** que significa "global regular expression print" n'és un exemple clar.

Per exemple,

```
grep -E [0-9]+ Test.java
```

Mostraria totes les línies del fitxer Test.java que continguin seqüències de números. No massa útil... Les línies amb noms de variables amb números, com ara x1, també es mostrarien.

Eliminem la possibilitat de mostrar números precedits immediatament per una lletra:

```
grep -E [^A-Za-z][0-9]+ Test.java
```

El conjunt [^A-Za-z] indica qualsevol lletra que **no** està dins dels rangs A-Z i a-z.

Per exemple, la següent expressió regular podria identificar correus electrònics:

```
\b[A-Za-z0-9]+@[A-Za-z0-9]+\.[A-Za-z]{2,6}\b
```

Les expressions regulars són complexes i dependents del llenguatge i queden fora d'aquest curs.

66.9. Classe Scanner i la codificació de caràcters

Al crear un objecte Scanner es pot indicar la codificació de caràcters utilitzada per interpretar la lectura de les dades.

Per exemple:

```
Scanner in = new Scanner(System.in, "Windows-1252");
```

Només cal indicar el nom de la codificació com a segon paràmetre a la creació de l'objecte.

66.10. Llegir un fitxer de text caràcter a caràcter

A vegades voldrem llegir un fitxer caràcter a caràcter, en aquets cas tenim dues opcions.

1. Prenem tota la línia amb el mètode **nextLine()** de la classe Scanner i manipulem la línia amb els mètodes de les classes **String** i **Character**.
2. Establim el mètode **useDelimiter** de la classe Scanner amb una cadena buida.

```
Scanner in = new Scanner(.....);
in.useDelimiter("");
```

66.11. Classificar caràcters

Pot ser interessant recordar alguns dels mètodes de la classe **Character** que permeten distingir un tipus de caràcter d'un altre.

isDigit, **isLetter**, **isUpperCase**, **isLowerCase**, **isWhiteSpace**, ...

66.12. Llegir un fitxer de text línia a línia

Els mètodes **substring** i **trim** de la classe String poden ser útils per fer un tractament de les dades d'un fitxer de text línia a línia.

66.13. Escanejar una cadena

Es pot utilitzar la classe **Scanner** en una variable cadena.

```

linia = "Pere Lòpez Antuno 8 Maria Ferrer Ortuño 5 Eva Gusi Pons";
Scanner scannerLinia = new Scanner(linia);
String nomComplert = "";
while (!scannerLinia.hasNextInt()) {
    nomComplert = nomComplert + " " + scannerLinia.nextInt();
}

```

66.14. Com obrir un PrintWriter per afegir dades

Per defecte, quan s'accedeix a un fitxer de text mitjançant un objecte PrintWriter si aquest conté informació s'elimina permanentment.

A vegades interessa accedir a un fitxer de text amb la intenció d'afegir-hi dades sense modificar-ne les existents.

Algunes classes dedicades a l'escriptura de fitxers permeten indicar si el què és pretén és afegir dades o bé crear el fitxer de nou. En general això s'indica passant un valor booleà **true** com a segon paràmetre a l'hora de crear l'objecte, per exemple, una d'aquestes classes és **FileOutputStream**:

```

FileOutputStream fos = new FileOutputStream(file, true); // Obre el
fitxer en modalitat append

```

El problema és que la classe **PrintWriter** no permet aquest segon paràmetre. No obstant si que es poden crear objectes d'aquest tipus proporcionant un objecte **FileInputStream** durant la seva creació, a més, si aquest objecte està preparat per treballar en modalitat **append** també ho estarà el nou **Printwriter** creat. Per exemple:

```

File file = new File("nomFitxer");
PrintWriter pw = null;
try {
    FileOutputStream fos = new FileOutputStream(file, true);
    pw = new PrintWriter(fos);
    // Afegim dades al fitxer

} catch (FileNotFoundException ex) {
    ex.printStackTrace();
} finally {
    pw.close();
}

```

```
// Al tancar el PrintWriter es tanca automàticament el  
FileInputStream  
}
```

Lectura i escriptura en fitxers binaris d'accés aleatori

En algunes situacions, llegir un fitxer de forma seqüencial, de principi a fi, pot ser poc eficient.

Els fitxers d'accés aleatori permeten accedir directament a localitzacions arbitràries en un fitxer de disc.

Són molt útils quan es vol emmagatzemar registres de dades en una estructura de taula.



L'accés aleatori **només es suportat pels fitxers en un disc.**

Els fluxes **System.in** i **System.out** associats al teclat i a la finestra terminal no suporten aquest tipus d'accés.

Cada fitxer de disc manté un apuntador amb una adreça de memòria que indica la posició actual de lectura/escriptura del fitxer. Normalment, aquest apuntador es troba al final del fitxer i per tant en general s'afegeixen dades al final.

En Java existeix la classe **RandomAccessFile** que proporciona els mètodes necessaris per accedir a un fitxer i gestionar-ne l'apuntador manualment. Aquest tipus d'accés a un fitxer s'anomena **accés aleatori**.

Si es mou aquest apuntador a una altra posició del fitxer, la lectura o l'escriptura del fitxer es realitzarà precisament en el punt indicat per l'apuntador, a més, cada

cop que es fa una operació de lectura o escriptura, l'apuntador es desplaçarà automàticament el mateix nombre de bytes amb què s'ha operat.

El nombre de bytes desplaçats dependrà de la mida associada al tipus primitiu del Java que s'ha escrit. Cal tenir sempre en compte que, com que RandomAccessFile treballa amb fitxers orientats a byte, aquesta classe transforma el valor del tipus primitiu a una seqüència de bytes, segons el mètode invocat.

Per saber exactament en quant ha variat l'apuntador del fitxer, cal saber quants bytes ocupa cada tipus de dades involucrat, cosa **fàcil pels tipus primitius però delicada pels String**, per exemple.

Nombre de bytes escrits per a cadascun dels mètodes write de la classe RandomAccessFile

Mètode	bytes
writeBoolean(boolean b)	1
writeByte(byte v)	1
writeChar(char c)	2
writeDouble(double d)	8
writeFloat(float f)	4
writeInt(int i)	4
writeLong(long l)	8
writeShort(short s)	2

67.1. Escriptura seqüencial sobre un fitxer aleatori

1. Creem el fitxer i donem permisos d'accés

```
File f = new File( . . . );
```

```
RandomAccessFile raf = new RandomAccessFile(f, "rw");
```

En aquest moment el fitxer ocupa 0 bytes.

2. Escrivim un enter al fitxer.

```
raf.writeInt(2);
```

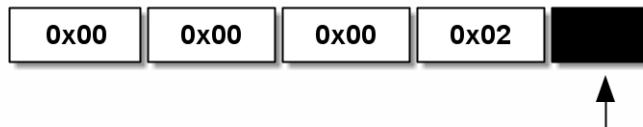


Figura 67.1. Contingut del fitxer

El fitxer ocupa 4 bytes, la fletxa indica la següent posició d'escriptura.

3. Escrivim un altre enter al fitxer.

```
raf.writeInt(10);
```

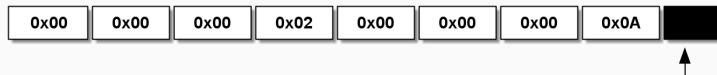


Figura 67.2. Contingut del fitxer

El fitxer ocupa 4 bytes, la fletxa indica la següent posició d'escriptura.

67.2. Classe RandomAccessFile

Per obrir un fitxer d'accés aleatori cal proporcionar el **nom** del fitxer i una cadena especificant el **tipus d'accés**, lectura (**r**), escriptura (**w**) o lectura-escriptura (**rw**).

Per exemple:

```
RandomAccessFile f = new RandomAccessFile("bank.dat", "rw");
```

El mètode **seek** mou l'apuntador a la posició indicada contant des del principi del fitxer.

```
f.seek(posicio);
```

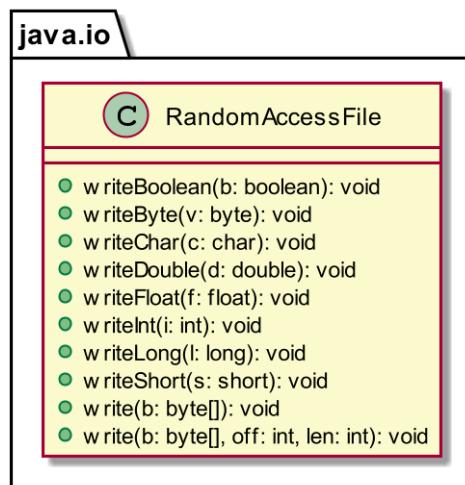
El primer byte del fitxer té la posició 0. Per trobar la posició actual del apuntador s'utilitza el mètode **getFilePointer(): long**

```
long posicio = f.getFilePointer()
```

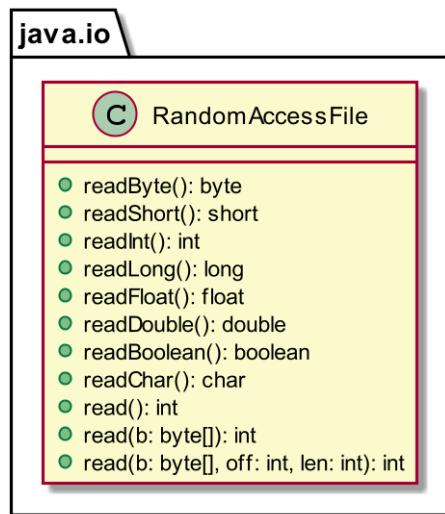
Per determinar la longitud, en bytes, del fitxer s'utilitza el mètode **length** .

```
long longitudFitxer = f.length();
```

Per escriure dades es pot utilitzar qualsevol dels mètodes write de la classe.



Per llegir dades es pot utilitzar qualsevol dels mètodes read de la classe.



Per exemple,

Aquest programa escriu una seqüència de 20 valors enters.

```
import java.io.File;
import java.io.RandomAccessFile;
import java.io.FileNotFoundException;
import java.io.IOException;

public class EscriureEntersDoblesBinari {
    public static void main(String[] args) {
        try {
            RandomAccessFile raf = null;
            try {
                File f = new File("Enters.bin");
                raf = new RandomAccessFile(f, "rw");

                //Ara no hi ha delimitadors. S'escriuen els valors
                consecutius.
                //Es van generant els valors i escrivint
                int valor = 1;
                for (int i = 0; i < 20; i++) {
                    raf.writeInt(valor);
                    valor = valor * 2;
                }

                System.out.println("Fitxer escrit satisfòriament.");
            } catch (IOException e) {
                e.printStackTrace();
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

```
// La mida d'un enter són 4 bytes.  
// La mida del fitxer hauria de ser 20*4 = 80 bytes  
} finally {  
    raf.close();  
}  
} catch (FileNotFoundException ex) {  
    System.err.println("No s'ha trobat el fitxer: " + ex);  
} catch (IOException ex) {  
    System.err.println("Error escrivint dades: " + ex);  
}  
}  
}  
}
```

Aquest programa llegeix una seqüència de 20 enters.

```
import java.io.File;  
import java.io.RandomAccessFile;  
import java.io.FileNotFoundException;  
import java.io.IOException;  
  
public class LlegirEntersBinari {  
    public static void main (String[] args) {  
        try {  
            RandomAccessFile raf = null;  
            try {  
                File f = new File("Enters.bin");  
                raf = new RandomAccessFile(f, "r");  
  
                long numEnters = f.length() / 4;  
                System.out.println("Hi ha " + numEnters + " enters.");  
                for (int i = 0; i < numEnters; i++) {  
                    int valor = raf.readInt();  
                    System.out.println("S'ha llegit el valor " + valor);  
                }  
            } finally {  
                raf.close();  
            }  
        } catch (FileNotFoundException ex) {  
            System.err.println("No s'ha trobat el fitxer: " + ex);  
        } catch (IOException ex) {  
            System.err.println("Error escrivint dades: " + ex);  
        }  
    }  
}
```

Aquest programa sobreescriu els cinc primers enters dels exemples anteriors.

```
import java.io.File;
import java.io.RandomAccessFile;
import java.io.FileNotFoundException;
import java.io.IOException;

public class SobreescrivireEntersBinari {
    public static void main (String[] args) {
        try {
            RandomAccessFile raf = null;
            try {
                File f = new File("Enters.bin");
                raf = new RandomAccessFile(f, "rw");

                //L'apuntador està al primer byte
                long apuntador = raf.getFilePointer();
                System.out.println("Inici: Apuntador a posició " +
apuntador);

                //Sobreescrivim els primers cinc valors
                for (int i = 0; i < 5; i++) {
                    raf.writeInt(-1);
                }

                //Si el fitxer ja tenia més de cinc enters, al final hi
ha brossa
                apuntador = raf.getFilePointer();
                System.out.println("Fi: Apuntador a posició " +
apuntador);

                //Es fixa la mida del fitxer als valors escrits
                raf.setLength(apuntador);

                System.out.println("Fitxer modificat correctament.");
            } finally {
                raf.close();
            }
        } catch (FileNotFoundException ex) {
            System.err.println("No s'ha trobat el fitxer: " + ex);
        } catch (IOException ex) {
            System.err.println("Error escrivint dades: " + ex);
        }
    }
}
```

```
}
```

Lectura / escriptura d'un fitxer de text.

```
package provesfitxers;

import java.io.IOException;
import java.io.RandomAccessFile;

public class RandomAccessFileEx {

    static final String FILENAME = "./src/provesfitxers/input.txt";

    public static void main(String[] args) {

        try {
            System.out.println(new
String(readFromFile(FILENAME, 150, 15)));
            writeToFile(FILENAME, "Cadena de Prueba", 150);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private static byte[] readFromFile(String filePath, int
position, int size) throws IOException {
        RandomAccessFile file = null;
        byte[] bytes;
        try {
            file = new RandomAccessFile(filePath, "r");
            file.seek(position);
            bytes = new byte[size];
            file.read(bytes);
        } finally {
            file.close();
        }
        return bytes;
    }

    private static void writeToFile(String filePath, String data, int
position) throws IOException {

        RandomAccessFile file = null;
        try {
            file = new RandomAccessFile(filePath, "rw");

```

```

        file.seek(position);
        file.write(data.getBytes());
    } finally {
        file.close();
    }
}
}

```

67.3. Mètodes de la classe RandomAccessFile

java.io

C RandomAccessFile

- RandomAccessFile(file: File, mode: String)
- RandomAccessFile(name: String, mode: String)
- close(): void
- getFilePointer(): long
- setLength(size: long)
- length(): long
- read(): int
- read(b: byte[]): int
- read(b: byte[], off: int, len: int): int
- readT(): T on T és un tipus
- seek(pos: long): void
- setLength(new Length: long): void
- skipBytes(int n): int
- write(b: byte[]): void
- write(b: byte[], off: int, len: int): void
- write(t: T) on T és un tipus

RandomAccessFile(file: File, mode: String)

Crea un objecte RandomaccessFile a partir de l'objecte File especificat i un mode d'accés.

RandomAccessFile(name: String, mode: String)

Crea un objecte RandomaccessFile a partir del seu nom i un mode d'accés.

close(): void

Tanca el flux de dades i allibera tots els recursos associats.

getFilePointer(): long

Retorna la posició en bytes, des del principi del fitxer, on es produirà la següent operació de lectura/escriptura.

setLength(size: long): void

Modifica la mida del fitxer. Si la mida és més gran que l'actual, s'afegeixen zeros.

length(): long

Retorna el nombre de bytes del fitxer.

read(): int

Llegeix un byte de dades del fitxer. Retorna -1 al quan arriba al final del flux de dades.

read(b: byte[]): int

Llegeix fins a b.length bytes de dades del fitxer a una matriu de bytes.

read(b: byte[], off: int, len: int): int

Llegeix fins a **len** bytes de dades des del fitxer a una matriu de bytes.

readT(): T on T és un tipus

Diferents versions del mètode en funció del tipus de dades.

seek(pos: long): void

Estableix la posició en bytes, especificada per **pos**, des del principi del fitxer on es produirà la següent operació de lectura/escriptura.

skipBytes(int n): int

Salta **n** bytes d'entrada.

write(b: byte[]): void

Escriu **b.length** bytes de la matriu especificada al fitxer començant al punt on apuntador del fitxer.

write(b: byte[], off: int, len: int): void

Escriu **len** bytes de la matriu especificada al fitxer començant a la posició **off** del fitxer.

write(t: T) on T és un tipus

Diferents versions del mètode en funció del tipus de dades.

Expressions regulars

Sovint és necessari escriure codi per validar dades entrades per part de l'usuari, adreces de correu, números de la seguretat social, etc...

Una manera senzilla de realitzar aquestes validacions és utilitzant expressions regulars.

Una expressió regular, abreujadament regex, és una cadena que descriu un patró que compleixen un conjunt de cadenes. Es poden utilitzar expressions regulars per validar, reemplaçar i dividir cadenes.

Les classes que ens permeten treballar amb les expressions regulars es troben al paquet **java.util.regex**

68.1. Sintaxi de les expressions regulars

Una **expressió regular** consisteix en **literals caràcters i símbols especials**, la següent taula resumeix les expressions regulars utilitzades més freqüentment.



Un caràcter espai en blanc ' ' equival a '\t', '\n', '\r' o bé '\f'. Per tant, l'expressió \s és la mateixa que [\t\n\r\f], i l'expressió \S és la mateixa que [^ \t\n\r\f].

Taula 68.1. Expressions regulars comuns

Expressió regular	Significat	Exemple
x	un caràcter concret x	Java coincideix amb Java
.	un únic caràcter	Java coincideix amb J..a

Expressió regular	Significat	Exemple
(ab cd)	ab o cd	Java coincideix amb J(av uli)
[abc]	a, b, o c	Java coincideix amb J[abcde]va
[^abc]	un caràcter excepte a, b o c	Java coincideix amb Ja[^ abcde]a
[a-z]	de la a a la z	Java coincideix amb [A-N]av[a-u]
[^a-z]	qualsevol caràcter excepte de la a a la z	Java coincideix amb Jav[^b-z]
[a-e[m-p]]	de la a a la e o bé de la m a la p	Java coincideix amb [A-G[I-M]]av[a-d]
[a-e&&[c-p]]	la intersecció de a-e amb c-p	Java coincideix amb A-P&&[I-M]av[a-d]
\d	un dígit, equivalent a [0-9]	Java2 coincideix amb "Java[\d]"
\D	un no dígit	\$Java coincideix amb "[\D][\D]ava"
\w	una lletra ^a	Ja_va3 coincideix amb "[\w]a[\w]va[\w]"
\W	una no lletra ^b	\$Java coincideix amb "[\W]Java"
\s	un espai en blanc	Java 2 coincideix amb "Java\s2"
\S	un no espai en blanc	Java coincideix amb "[\S]ava"
p*	zero o més ocurredades del patró p	aaaabb coincideix amb a*bb ababab coincideix amb (ab)*
p+	una o més ocurredades del patró p	a coincideix amb a+b abcd coincideix amb (ab)+.*
p?	zero o una ocurredade del patró p	Java coincideix amb J?Java Java coincideix amb J?ava
p{n}	exactament n ocurredades del patró p	Java coincideix amb Ja{1}.* Java no coincideix amb .{2}

Expressió regular	Significat	Exemple
$p\{n,\}$	com a mínim n ocurrències del patró p	aaaa coincideix amb a{1,} a no coincideix amb a{2,}
$p\{n,m\}$	entre n i m ocurrències del patró p (inclou extrems)	aaaa coincideix amb a{1,9} abb no coincideix amb a{2,9}bb
$^$	Busca coincidències al principi d'una línia	^abc coincideix amb abc ^abc no coincideix amb aaa abc
$$$	Busca coincidències al final d'una línia	\$abc coincideix amb aaa abc \$abc no coincideix amb abc aaa

a Quan parlem de **lletra** ens referim a una lletra, un dígit o el guió baix.

b Quan parlem de **lletra** ens referim a una lletra, un dígit o el guió baix.



Quan parlem de **lletra** ens referim a una lletra, un dígit o el guió baix.

Per tant **\w** és el mateix que **[a-zA-Z][0-9]_** o simplement **[a-zA-Z0-9_]**, i **\W** és el mateix que **[^a-zA-Z0-9_]**.



No deixar espais quan s'utilitzen quantificadors, és a dir, *, +, ?, {n}, {n,}, i {n,m}.

Per exemple el patró **a{2, 6}** és incorrecte perquè hi ha un espai després de la coma.



Cal utilitzar parèntesis per agrupar els patrons.

Per exemple, **(ab){3}** coincideix amb **ababab** però **ab{3}** coincideix amb **abbb**.

68.2. Algunes expressions regulars d'exemple

[a-z]{3,6}[ts]

Qualsevol lletra minúscula de l'abecedari en grups d'entre tres a sis lletres, seguit d'un caràcter "t" i/o "s".

[^q][a-z]{2}[]

Qualsevol grup de dues minúscules que no estiguin precedides per una “q” i estiguin seguides per un espai.

^[a-zA-Z0-9.!#\$%&'^=?`{|}~-]@[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?(?:\.[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?)\$

Una adreça email

^(0?[1-9]|1[2][0-9]|3[01])[\\/-](0?[1-9]|1[012])[\\/-]\\d{4}\$

Una data

(\\d{4})([BCDFGHJKLMNOPRSTVWXYZ]{3})

Una matrícula d'Espanya

68.3. Treballar amb expressions regulars

Java proporciona dues maneres de treballar amb les expressions regulars:

1. Aplicant-les directament sobre objectes **String**.
2. Utilitzant classes específiques per la manipulació de les expressions regulars.

68.3.1. Aplicar expressions regulars sobre objectes String

La classe **String** té un mètode anomenat **matches(String regex)** que retorna **true** si una cadena compleix una expressió regular.

Vegem-ho amb alguns exemples:

Exemple 1

```
import java.util.Scanner;

public class TestRegex {

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);

        System.out.print("Segur que cols sortir de l'aplicació? ");
        String sortir = in.nextLine();
        while (!sortir.toLowerCase().matches("s|n|si|no|sí")) {
```

```
System.out.println("No entenc la resposta");
System.out.print("Segur que vols sortir de l'aplicació? ");
sortir = in.nextLine();
}
System.out.println("adéu");
}
}
```

Exemple 2

Format de telèfon fix.

```
System.out.println("(97) 761 23 33".matches("\\"([1-9]\\d\\) [\\d]{3}
[\\d]{2} [\\d]{2}")); // true
System.out.println("(07) 761 23 33".matches("\\"([1-9]\\d\\) [\\d]{3}
[\\d]{2} [\\d]{2}")); // false
```

Exemple 3

Detectar números parells.

```
System.out.println("224".matches("[\\d]*[02468]")); // true
System.out.println("123".matches("[\\d]*[02468]")); // false
```



Per tractar (i) com a caràcters s'han d'escapar donat que tenen seignificant dins de les expressions regulars.

Exemple 4

Validar cognoms.

```
System.out.println("López".matches("[A-ZçàáèéíòóúÀÈÉÍÒÓÚ][a-zA-
ZçàáèéíòóúÀÈÉÍÒÓÚ]{1,24}")); // true
System.out.println("lópez".matches("[A-ZçàáèéíòóúÀÈÉÍÒÓÚ][a-zA-
ZçàáèéíòóúÀÈÉÍÒÓÚ]{1,24}")); // false
System.out.println("López123".matches("[A-ZçàáèéíòóúÀÈÉÍÒÓÚ][a-zA-
ZçàáèéíòóúÀÈÉÍÒÓÚ]{1,24}")); // false
```

68.3.2. Altres mètodes regex de la classe String

C	String
●	matches(regex: String): boolean
●	replaceAll(regex: String, replacement: String): String
●	replaceFirst(regex: String, replacement: String): String
●	split(regex: String): String[]
●	split(regex: String, limit: int): String[]

68.4. Classes específiques per les expressions regulars

Una manera més sofisticada de treballar amb expressions regulars és usant les classes **Pattern** i **Matcher**, incloses dins el package **java.util.regex**, que han estat creades amb aquest propòsit específic.

La primera serveix per establir una expressió regular (concretament per **compilar-la**), mentre que la segona és l'encarregada de **processar-la** i indicar si troba coincidències.

La principal particularitat d'aquestes classes és que **permeten cercar coincidències ja siguin parcials o totals dins la cadena de text**, així com identificar exactament on han succeït. Això atorga un nivell de detall molt superior al que proporcionen els mètodes de la classe String.

El procés per usar aquestes dues classes sempre és el següent:

1. Instanciar un objecte **Pattern** usant el seu mètode estàtic **compile(String regex)**.
2. Obtenir un objecte **Matcher** associat al text a processar, invocant **matcher(CharSequence input)**.
3. A partir de l'objecte Matcher, es pot processar el text de diverses maneres invocant els diferents mètodes que ofereix aquesta classe.

68.4.1. La classe Pattern

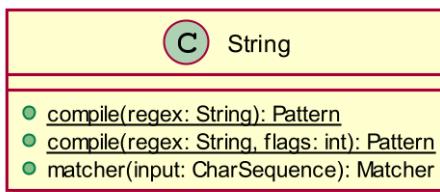


Figura 68.1. Principals mètodes de la classe Pattern

compile(regex: String): Pattern

Compila l'expressió regular i retorna un Pattern. Fa la funció del constructor del Pattern.

compile(regex: String, flags: int): Pattern

Compila l'expressió regular i retorna un Pattern. Fa la funció del constructor del Pattern.

Permet indicar diferents flags per modificar el comportament de les coincidències amb l'expressió regular. Per exemple:

Pattern.CASE_INSENSITIVE no té en compte les majúscules i les minúscules alhora de comprovar una coincidència amb l'expressió regular.

matcher(input: CharSequence): Matcher

Crea un objecte **Matcher** contra l'expressió continguda al **Pattern**.

68.4.2. La classe Matcher

La classe **Matcher** permet anar cercant coincidències dins la cadena de text, una a una. Els seus mètodes més rellevants són:

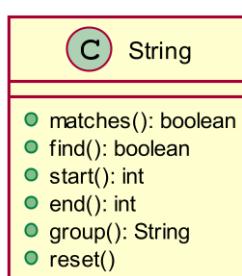


Figura 68.2. Principals mètodes de la classe Matcher

matches(): boolean

Intenta validar tota la cadena contra el patró.

find(): boolean

Cerca si hi ha alguna coincidència dins la cadena de text a processar. Successives invocacions van retornant **true** mentre es trobi alguna coincidència. Un cop s'han esgotat totes les coincidències possibles, retorna **false**.

start(): int

Indica quin és l'índex del caràcter on s'inicia la darrera coincidència trobada (el darrer cop que s'ha cridat **find()** i ha retornat **true**).

end(): int

Indica quin és l'índex del primer caràcter posterior a la darrera coincidència.

group(): String

Indica quina és la cadena de text exacta que ha provocat la darrera coincidència.

reset()

Reinicialitza l'objecte, de manera que es torna a començar des de zero el procés de detecció de coincidències.

68.5. Alguns exemples

68.5.1. Exemple 1

```
import java.util.regex.Pattern;

public class TestRegex {

    public static void main(String[] args) {
        String regex = "\\d";
        String text = "un4dos6tres7quatre9cinc";

        Pattern p = Pattern.compile(regex);
        String[] items = p.split(text);
        for (String s : items) {
            System.out.println(s);
        }
    }
}
```

```

    }
}
```

```

un
dos
tres
quatre
cinc
```

68.5.2. Exemple 2

```

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class TestRegex {

    public static void main(String[] args) {

        String text = "El color groc és quadrat o rodó?";
        String regex = "[a-z]{2,5}[d]"; // Cadenes d'entre 2 i 5 lletres
        seguides d'una d o una c

        Pattern p = Pattern.compile(regex);
        Matcher m = p.matcher(text);
        while (m.find() == true) {
            System.out.print("Cadena: " + m.group());
            System.out.print(" (Inici: " + m.start());
            System.out.println(", Fi: " + m.end() + ")");

        }
    }
}
```

```

Cadena: groc (Inici: 9, Fi: 13)
Cadena: quad (Inici: 17, Fi: 21)
Cadena: rod (Inici: 27, Fi: 30)
```

68.5.3. Exemple 3

```

import java.util.regex.Matcher;
import java.util.regex.Pattern;
```

```

public class TestRegex {

    public static void main(String[] args) {
        String regex = "\bgos\b"; // \b detecta la vora d'una paraula,
        \B just el contrari
        String text = "gos gosset gossera gos agosarat";

        Pattern p = Pattern.compile(regex);
        // get a matcher object
        Matcher m = p.matcher(text);
        int count = 0;
        while (m.find()) {
            count++;
            System.out.println("Match number " + count);
            System.out.println("start(): " + m.start());
            System.out.println("end(): " + m.end());
        }
    }
}

```

```

Match number 1
start(): 0
end(): 3
Match number 2
start(): 19
end(): 22

```

68.5.4. Exemple 4

```

import java.util.Scanner;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class TestRegex {

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        String regex = "([0-9]{2})/([0-9]{2})/([0-9]{4})";

        Pattern p = Pattern.compile(regex);
        Matcher m;
        String dia = "";

```

Exemple 4

```
String mes = "";
String any = "";
do {
    System.out.print("Entra una data (dd/mm/yyyy): ");
    String data = in.nextLine();
    m = p.matcher(data);
    if (m.matches()) {
        // m.group(0) és tota l'expressió
        dia = m.group(1);
        mes = m.group(2);
        any = m.group(3);
    }
} while (!m.matches());

System.out.println(mes + "/" + dia + "/" + any);
}
```

Bibliografia

Llibres

Kishori Sharan. Java Language Features: With Modules, Streams, Threads, I/O, and Lambda Expressions. Apress. 2018. ISBN 978-1-484-23347-4

Ralph Lecessi. Functional Interfaces in Java. Apress. 2019. ISBN. 978-1-4842-4277-3

Herbert Schildt. Java: The Complete Reference, Eleventh Edition. McGraw Hill Professional. 2018. ISBN 978-1-260-44024-9

Y. Daniel Liang. Intro to Java Programming. 10th ed. Pearson 2014. ISBN 978-0-133-76131-3

Web

Oracle Java Documentation. The Java Tutorials. <https://docs.oracle.com/javase/tutorial/index.html>

Part VI. UF6 POO. Introducció a la persistència en les BD

Sumari

69. Resultats d'aprenentatge i criteris d'avaluació	833
70. Continguts	835
71. Introducció a JDBC	837
71.1. ODBC - Open DataBase Connectivity	837
71.2. JDBC - Java Database Connectivity	838
71.3. Drivers JDBC	839
71.3.1. Tipus de drivers JDBC	839
72. Instal·lació del MySQL/MariaDB	841
72.1. Instal·lació de MySQL/MariaDB en Debian	841
72.2. Instal·lació de MySQL/MariaDB en Windows	842
72.3. Importació de bases de dades d'exemple	842
72.4. Creació d'usuaris	843
72.4.1. Creació d'usuaris	843
72.4.2. Veure la llista d'usuaris creats	845
73. Establiment de connexions	847
73.1. Incorporació del driver al projecte	847
73.2. Prova de connexió	849
73.3. Formes alternatives d'establir la connexió	851
74. Sentències DML. Select	853
74.1. Obtenir els camps d'un registre a partir del nom del camp	853
74.2. Alguns tipus de dades poden presentar dificultats	854
74.3. Obtenir els camps d'un registre a partir de l'índex del camp	857
74.4. Seleccions parametritzades i injecció de codi	858
75. Sentències preparades	863
75.1. SQL-Injection	863
75.2. Ús de les sentències preparades	865
75.3. LIKE vs =	867
76. Sentències DML i DDL. Modificació de dades	869
76.1. Obtenció dels <i>id</i> autogenerats	871
77. Interfície Resultset	873
77.1. Tipus de resultsets	873
77.2. Concurrència d'un ResultSet	874
77.3. Mantenir el cursor en un ResultSet	875
77.4. Modificar dades a través d'un ResultSet	875

77.4.1. Modificar el valor d'una columna	875
77.4.2. Inserir un nou registre	876
77.4.3. Eliminar un registre existent	876
77.5. Un exemple sencer	877
78. Metadades	879
78.1. Mètode main	881
78.2. Obtenir informació de la base de dades	881
78.3. Obtenir informació de les taules de la base de dades	882
78.4. Obtenir informació d'una taula	884
78.5. Metadades de consultes	885
78.6. Sentències explain	887
79. Procediments emmagatzemats	891
79.1. Configuració de MariaDB	891
79.2. Crida a procediments	893
79.3. Crida a funcions	895
79.4. Crida a procediments des de Java	898
79.5. Crides a funcions des de Java	900
80. Transaccions	901
80.1. Autocommit	901
80.2. Exemple	902
Bibliografia	905

Resultats d'aprenentatge i criteris d'avaluació

1. Gestiona informació emmagatzemada en bases de dades relacionals mantenint la integritat i la consistència de les dades.
 1. Identifica les característiques i els mètodes d'accés a sistemes gestors de bases de dades relacionals.
 2. Programa connexions amb bases de dades.
 3. Escriu codis per emmagatzemar informació en bases de dades.
 4. Crea programes per recuperar i mostrar informació emmagatzemada en bases de dades.
 5. Efectua esborraments i modificacions sobre la informació emmagatzemada.
 6. Crea aplicacions que executin consultes sobre bases de dades.
 7. Crea aplicacions per possibilitar la gestió d'informació present en bases de dades relacionals.
2. Gestiona informació emmagatzemada en bases de dades objecte-relacionals mantenint la integritat i la consistència de les dades.
 1. Identifica les característiques de les bases de dades objecte-relacionals.
 2. Analitza la seva aplicació en el desenvolupament d'aplicacions mitjançant llenguatges orientats a objectes.

-
- 3. Classifica i analitza els diferents mètodes que suporten els sistemes gestors de bases de dades per a la gestió de la informació emmagatzemada de forma objecte-relacional.
 - 4. Programa aplicacions que emmagatzemem objectes en bases de dades objecte-relacionals.
 - 5. Realitza programes per recuperar, actualitzar i eliminar objectes de les bases de dades objecte-relacionals.
 - 6. Realitza programes per emmagatzemar i gestionar tipus de dades estructurades, compostos i relacionats.
3. Utilitza bases de dades orientades a objectes, analitzant-ne les característiques i aplicant tècniques per mantenir la persistència de la informació.
- 1. Identifica les característiques de les bases de dades orientades a objectes.
 - 2. Analitza la seva aplicació en el desenvolupament d'aplicacions mitjançant llenguatges orientats a objectes.
 - 3. Defineix les estructures de dades necessàries per emmagatzemar objectes.
 - 4. Classifica i analitza els diferents mètodes que suporten els sistemes gestors per gestionar la informació emmagatzemada.
 - 5. Programa aplicacions que emmagatzemin objectes en les bases de dades orientades a objectes.
 - 6. Realitza programes per recuperar, actualitzar i eliminar objectes de les bases de dades orientades a objectes.
 - 7. Realitza programes per emmagatzemar i gestionar tipus de dades estructurades, compostos i relacionats.

70

Continguts

1. Disseny de programes amb llenguatges de programació orientats a objectes per gestionar bases de dades relacionals:
 1. Establiment de connexions.
 2. Recuperació d'informació.
 3. Manipulació de la informació.
2. Disseny de programes amb llenguatges de programació orientats a objectes per gestionar bases de dades objecte-relacionals:
 1. Establiment de connexions.
 2. Recuperació d'informació.
 3. Manipulació de la informació.
3. Disseny de programes amb llenguatges de programació orientats a objectes per la gestió de bases de dades orientades a objectes:
 1. Introducció a les bases de dades orientades a objectes.
 2. Característiques de les bases de dades orientades a objectes.
 3. El model de dades orientat a objectes.
 - i. Relacions.
 - ii. Integritat de les relacions.
 - iii. UML.
 4. El model estàndard ODMG.

-
- i. Model d'objectes.
 - ii. Llenguatge de definició d'objectes ODL.
 - iii. Llenguatge de consulta d'objectes OQL.
5. Prototipus i productes comercials de SGBDOO.

71

Introducció a JDBC

Moltes aplicacions treballen amb dades persistents que, habitualment, s'emmagatzemem en una base de dades relacional.

El **Sistema Gestor de Bases de Dades** s'encarrega de resoldre les consultes, assegurar la integritat de les dades, i permetre l'accés concurrent a múltiples usuaris o instàncies de programes.

Això suposa una millora important respecte a la gestió de les dades a través de fitxers, especialment en els casos en què tinguem una quantitat gran de dades, amb una estructura interna més o menys complexa, i amb accessos concurrents.

Com a programadors, necessitarem que els nostres programes puguin comunicar-se amb el SGBD: establir una connexió, executar una sèrie de sentències de consulta i/o modificació de les dades emmagatzemades, i finalitzar la connexió.

Els sistemes de bases de dades més utilitzats són els relacionals, i el llenguatge per accedir a les dades el **SQL (Structured Query Language)**.

71.1. ODBC - Open DataBase Connectivity

L'**ODBC** és un estàndard de connexió a bases de dades relacionals, desenvolupat per Microsoft a principis dels 90, i convertit en un estàndard posteriorment.

L'**ODBC** utilitza un model de **tres capes** per accedir a la BD des d'una aplicació:

1. Per cada SGBD compatible amb ODBC existeix un **driver ODBC**. El driver ofereix una sèrie de funcionalitats d'accés al SGBD: tradueix les funcions que

utilitzem en el llenguatge de programació a les ordres concretes que s'han d'enviar al SGBD concret. Diferents drivers faran aquesta traducció a ordres diferents del SGBD, però mantindran una interfície gairebé idèntica de cara al llenguatge de programació.

2. Un **driver manager** s'encarregarà de mirar quins **drivers** tenim disponibles i de configurar-los per a poder-los utilitzar. També, el **driver manager** s'encarrega d'uniformar el procés de connexió a instàncies diferents del SGBD (per exemple, ens permet connectar a un servidor local, o a un servidor remot).
3. L'aplicació es comunica amb el **driver manager** per obtenir el driver adequat i establir una connexió amb la BD desitjada. Després, utilitza el **driver** per realitzar les seves consultes contra la BD.

71.2. JDBC - Java Database Connectivity

Una de les tasques que fa l'ODBC és traduir els tipus de dades SQL als tipus de dades del llenguatge de programació C.

Com que els tipus del C i els del Java són diferents, l'ODBC no és un sistema adequat per utilitzar directament des del Java.

A més, l'OBDC utilitza una interfície de programació C i és relativament difícil d'aprendre.

Per accedir a bases de dades relacionals, el Java proporciona les seves API pròpies com a part de l'edició estàndard, cosa que permet tenir una solució purament Java.

Ei JDBC funciona de manera similar a l'ODBC: existeixen **drivers** JDBC per a cadascun dels SGBD habituals i els utilitzem des de la nostra aplicació per accedir-hi.

A més, existeixen ponts ODBC-JDBC i a l'invers, que ens permeten accedir amb JDBC a sistemes que suportin ODBC però no JDBC.

Essencialment, utilitzarem JDBC per tres tasques:

- Establir una connexió al SGBD i BD.
- Enviar sentències SQL per tal que es processin al SGBD.

- Obtenir els resultats de les nostres sentències.

71.3. Drivers JDBC

Els drivers JDBC són biblioteques que permeten comunicar una aplicació Java amb una base de dades.

Tots els drivers JDBC segueixen les mateixes API: les especificades dins de Java SE. És a dir, el Java SE especifica una colla d'interfícies i els seus mètodes, i tots els drivers JDBC, que poden ser de diversos fabricants, tenen una implementació pròpia, però que segueix les especificacions de les interfícies de JDBC.

No tots els drivers tenen per què tenir tota la funcionalitat especificada per JDBC implementada, però sempre tindran la bàsica. Les interfícies principals que declara JDBC són **Statement**, **Connection**, **ResultSet**, **PreparedStatement** i **CallableStatement**.

Habitualment, utilitzarem els drivers implementats pel creador del SGBD que utilitzem.

71.3.1. Tipus de drivers JDBC

Depenent de la tecnologia utilitzada, els drivers JDBC es divideixen en diverses categories:

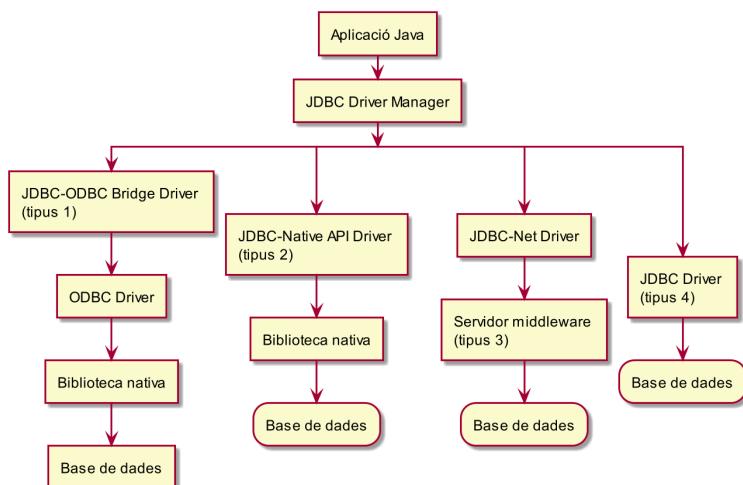


Figura 71.1. Arquitectura dels drivers JDBC

JDBC-ODBC Bridge Driver (Tipus 1)

Són drivers que actuen com a pont entre JDBC i ODBC. Tenen l'avantatge que amb un drivers de tipus 1 podem connectar amb qualsevol base de dades per a la qual ja existeixi un driver ODBC. A canvi, no és portable (perquè no és només Java), i cal tenir un driver ODBC instal·lat a la màquina per la versió exacte del SGBD que tinguem.

JDBC-Native API Driver (Tipus 2)

És una implementació mixta entre Java i codi natiu, que utilitza les funcions del sistema operatiu per a realitzar la connexió. És més ràpid que el tipus 1, però segueix sense una solució només Java, i necessitem tenir el driver natiu instal·lat a l'ordinador. Si necessitem molta rapidesa, un driver de tipus 2 pot ser la millor elecció.

JDBC-Net Driver (Tipus 3)

Consisteix d'un driver Java que s'instal·la al client, i que es comunica amb un servidor intermig (middleware) que és el que es comunica finalment amb el SGBD. Això fa que el driver sigui portable, que no depengui del fabricant, però a canvi de tenir una aplicació extra al servidor.

JDBC Driver (Tipus 4)

És un driver Java que es comunica directament amb el SGBD, sigui local o remot. És portable i no requereix de capes addicionals, però necessitem un driver diferent per cada SGBD. Aquest és el tipus de driver més comú.

72

Instal·lació del MySQL/MariaDB

Per practicar amb JDBC necessitem un SGBD relacional. Farem una instal·lació local de MariaDB en el nostre entorn de desenvolupament.

El **MySQL/MariaDB** és un dels SGBD més utilitzats, i és especialment adequat per aplicacions web que necessiten força rapidesa i no treballen amb un volum molt gran de dades.

MariaDB és un **fork** de MySQL amb un model de desenvolupament més obert que aquest últim, i que proporciona algunes característiques addicionals, bo i mantenint la compatibilitat amb **MySQL**.

Tant un com l'altre són programari lliure i són adequats per seguir els exemples i exercicis d'aquests apunts.

També necessitarem el driver JDBC de connexió a MySQL.

72.1. Instal·lació de MySQL/MariaDB en Debian

1. Instal·lar el paquet **mariadb-server** i el paquet **mariadb-client**.
2. Un cop instal·lat, només podem connectar-nos al servidor amb l'usuari root del sistema, en local, i sense utilitzar contrasenya:

```
$ sudo mysql
```

```
$ su  
Contrasenya:  
# mysql
```

72.2. Instal·lació de MySQL/MariaDB en Windows

Podem utilitzar el [instal·lador de MySQL](#)¹, o l'[instal·lador de MariaDB](#)².

També podem utilitzar paquets integrats com el [XAMPP](#)³, que inclouen MariaDB, el servidor web Apache, el PHP i el Perl en un sol instal·lador.

La instal·lació per defecte del [XAMPP](#) està orientada a entorns de desenvolupament, així que no s'hauria d'utilitzar directament en entorns de producció (encara que es pot fer si es revisa la seva configuració).

72.3. Importació de bases de dades d'exemple

En aquest tema utilitzarem algunes BD que es troben disponibles a la web amb llicències lliures:

- Una [base de dades d'empleats](#)⁴. Aquesta és una base de dades amb poques taules, però molts registres.

Per instal·lar-la s'ha de baixar [el repositori corresponent a github](#)⁵. El més fàcil és baixar-ho clicant a [Download zip](#).

Després de descomprimir, podem passar-li l'script que genera la base de dades a mariadb:

```
$ cd <directori on hem descomprimit>
$ sudo mysql < employees.sql
```

O des de dins del client mysql:

```
$ cd <directori on hem descomprimit>
$ sudo mysql
mysql> source employees.sql
```

¹ <https://dev.mysql.com/downloads/installer/>

² <https://downloads.mariadb.org/mariadb/>

³ <https://www.apachefriends.org/download.html>

⁴ <https://dev.mysql.com/doc/employee/en/>

⁵ https://github.com/datacharmer/test_db

- [Sakila](#)⁶, una base de dades de lloguer de pel·lícules.

Aquesta base de dades és molt més completa a nivell de disseny i, a més de les taules i dades de prova, té vistes i procediments emmagatzemats.

Es pot baixar de la pàgina de [bases de dades d'exemple del MySQL](#)⁷.

Cal carregar primer el fitxer sakila-schema.sql, que crea l'estructura de la BD, i després el fitxer sakila-data.sql, que conté les dades.

72.4. Creació d'usuaris

Per crear un usuari de MySQL/MariaDB primer de tot ens hem de connectar com a **root**. Aquí ho farem des de línia de comandes, encara que també es pot fer això des d'entorn gràfic utilitzant programari addicional com el **MySQL Workbench** o el **PHPMyAdmin**.

Per connectar com a root farem:

```
vagrant@javamariadb:~$ sudo mariadb -u root
Welcome to the MariaDB monitor. Commands end with ; or \g.
Your MariaDB connection id is 31
Server version: 10.5.12-MariaDB-0+deb11u1 Debian 11

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear
the current input statement.

MariaDB [(none)]>
```

Després d'introduir la contrasenya hauríem de veure el **prompt** del SGBD.

72.4.1. Creació d'usuaris

A continuació podem crear un usuari amb la sentència SQL **CREATE USER** i assignar-li permisos amb **GRANT**, a continuació es mostren alguns exemples:

⁶ <https://dev.mysql.com/doc/sakila/en/>

⁷ <https://dev.mysql.com/doc/index-other.html>

Exemple 1

```
CREATE USER 'nom_usuari'@'localhost' IDENTIFIED BY 'contrasenya';
GRANT ALL PRIVILEGES ON nom_BD.* TO 'nom_usuari'@'localhost';
```

Això crearà un usuari que només es podrà connectar des de l'ordinador local i que tindrà tots els permisos sobre totes les taules de la base de dades especificada.



Cal executar el GRANT per cadascuna de les bases de dades que utilitzem.

Exemple 2

```
CREATE USER 'nom_usuari'@'%' IDENTIFIED BY 'contrasenya';
GRANT ALL PRIVILEGES ON *.* TO 'nom_usuari'@'%';
```

Això crearà un usuari que es podrà conectar des de qualsevol ip i tindrà permisos sobre a totes les bases de dades del servidor.



Per què l'exemple anterior funcioni correctament caldrà especificar a la configuració del servidor **des de quines adreces ip es permet l'accés**, per fer-ho cal afegir la següent línia al fitxer de configuració de mariadb.

Per exemple, en el cas d'un sistema debian afegiríem al fitxer **/etc/mysql/maria.conf.d** la següent línia si volem accedir al servidor de bases de dades des de qualsevol adreça ip.

```
bind-address = 0.0.0.0
```

Exemple 3

En entorns de producció es poden crear més usuaris amb permisos diferents, per aconseguir que el nostre programa tingui sempre els permisos mínims necessaris. Això ens ajuda a protegir les dades en cas que el nostre programa tingui una vulnerabilitat o s'hagi comès un error.

També pot resultar una bona idea crear un usuari administrador diferent de root, que puguem utilitzar directament:

```
sudo /usr/bin/mysql -e "GRANT ALL ON *.* TO 'nom_usuari'@'localhost'
IDENTIFIED BY 'contrasenya' WITH GRANT OPTION"
```

Aquest últim usuari ens permetrà administrar el MariaDB des d'entorns gràfics, com el Workbench o el PHPMyAdmin.



La clàusula **WITH GRANT OPTION** indica que l'usuari que s'acaba de crear té permisos per proporcionar accés als usuaris a les bases de dades del servidor.

72.4.2. Veure la llista d'usuaris creats

Podem veure els usuaris creats fent una consulta a la taula **mysql.user**

```
MariaDB [(none)]> select host,user,password from mysql.user;
+-----+-----+-----+
| Host      | User       | Password          |
+-----+-----+-----+
| localhost | mariadb.sys |                   |
| localhost | root        | invalid           |
| localhost | mysql       | invalid           |
| %          | admin        | *A4B6157319038724E3560894F7F932C8886EBFCF |
+-----+-----+-----+
4 rows in set (0.002 sec)
```


73

Establiment de connexions

73.1. Incorporació del driver al projecte

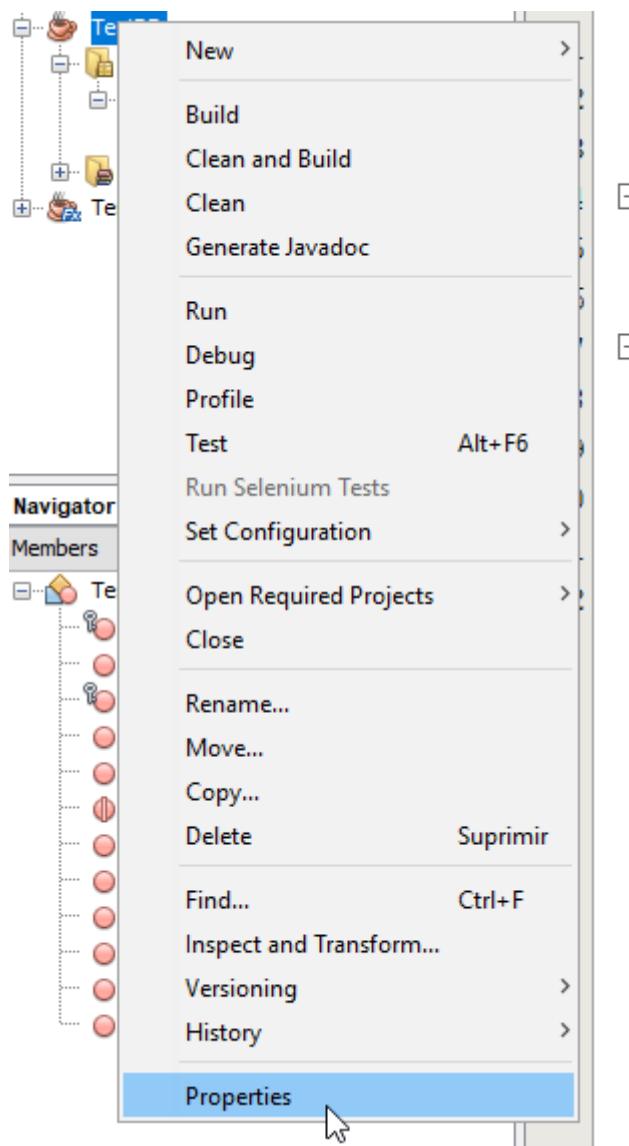
Per poder establir una connexió a MySQL/MariaDB des de Java, primer de tot cal incorporar el **driver** al nostre projecte de netbeans.



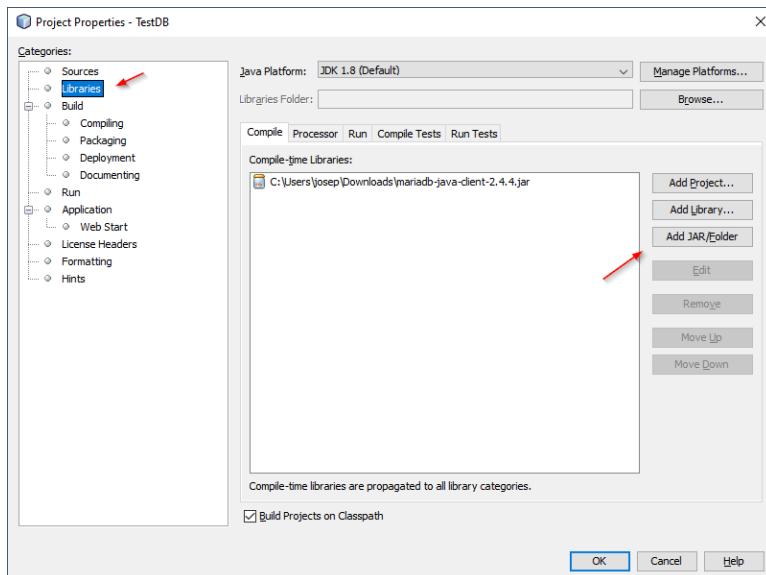
La incorporació del driver al projecte es pot automatitzar utilitzant un sistema com **Maven** o **Gradle**.

El connector per MariaDB es pot descarregar de <https://downloads.mariadb.org/connector-java/>.

En qualsevol dels dos casos hem d'acabar amb un fitxer **.jar**. Un cop tinguem el fitxer **.jar** el podem incorporar al projecte fent botó dret al projecte → Propietats.



I afegim el **.jar** al projecte:



73.2. Prova de connexió

Intentarem ara establir una connexió amb el servidor.

Les classes que apareixen als següents exemples s'han d'importar dels paquets **java.sql** i **javax.sql**. En aquests paquets es defineixen les interfícies que han de complir els **drivers** i cada **driver** proporciona les seves implementacions pròpies.

El següent exemple estableix una connexió contra el servidor local:

```

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class TestDB {

    public static void main(String[] args) {
        Connection con = null;
        try {
            con =
                DriverManager.getConnection("jdbc:mariadb://172.16.0.100:3306/sakila?
user=admin&password=1234");
            System.out.println("Connexió exitosa a la base de dades!");
        } catch (SQLException e) {
    }
}

```

```

        System.err.println("Error d'establiment de connexió: " +
e.getMessage());
    } finally {
        if (con != null) {
            try {
                con.close();
            } catch (SQLException ex) {
                System.err.println("Error de tancament de connexió:
" + ex.getMessage());
            }
        }
    }
}
}

```

O utilitzant un **try** amb recursos:

```

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class TestDBv1 {

    public static void main(String[] args) {
        try (Connection con =
DriverManager.getConnection("jdbc:mariadb://172.16.0.100:3306/sakila?
user=admin&password=1234")) { ❶
            System.out.println("Connexió exitosa a la base de dades!");
        } catch (SQLException e) {
            System.err.println("Error d'establiment de connexió: " +
e.getMessage());
        }
    }
}

```

Mereix especial atenció la línia de connexió:

```
jdbc:mariadb://172.16.0.100:3306/sakila?user=admin&password=1234
```

- En primer lloc apareix el protocol. Aquí hem de posar `jdbc:mariadb` si ens connectem a un servidor MariaDB, o `jdbc:mysql` si ens connectem a MySQL.

- Després tenim l'adreça on ens connectem, en el nostre cas es tracta d'un servidor de base de dades extern amb ip **172.16.0.100**, al port **3306**, que és el port per defecte de MySQL/MariaDB.
- A continuació tenim la base de dades que volem utilitzar, a l'exemple **sakila**.
- Després passem l'usuari i la contrasenya que utilitzem per establir la connexió.



Noteu que el format de la línia de connexió és el mateix que el que s'utilitza per indicar una URL al navegador web.

73.3. Formes alternatives d'establir la connexió

El mètode **DriverManager.getConnection()** està sobrecarregat i admet diversos formats per indicar els paràmetres de connexió.

En aquest exemple passem directament els paràmetres:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class TestDB2 {

    public static void main(String[] args) {
        String url = "jdbc:mariadb://172.16.0.100:3306/sakila";
        String user = "admin";
        String passwd = "1234";
        try (Connection con = DriverManager.getConnection(url, user,
                passwd);) { ①
            System.out.println("Connexió exitosa a la base de dades!");
        } catch (SQLException e) {
            System.err.println("Error d'establiment de connexió: " +
                    e.getMessage());
        }
    }
}
```

I en aquest últim exemple passem els paràmetres a través d'un objecte **Properties**. La classe **Properties** és un **Map** que s'utilitza per guardar parelles de **String**. S'utilitza sobretot per emmagatzemar les opcions de configuració d'un programa a un fitxer.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Properties;

public class TestDB3 {

    public static void main(String[] args) {

        String url = "jdbc:mariadb://172.16.0.100:3306/sakila";
        Properties propietatsConnexio = new Properties(); ①
        propietatsConnexio.setProperty("user", "admin"); ②
        propietatsConnexio.setProperty("password", "1234"); ③
        try (Connection con = DriverManager.getConnection(url,
propietatsConnexio);) { ④
            System.out.println("Connexió exitosa a la base de dades!");
        } catch (SQLException e) {
            System.err.println("Error d'establiment de connexió: " +
e.getMessage());
        }
    }
}
```

Sentències DML. Select

74.1. Obtenir els camps d'un registre a partir del nom del camp

Un cop hem establert la connexió a la base de dades podem començar a enviar-hi sentències SQL.

El següent exemple mostra per pantalla algunes dades dels 10 primers empleats (alfabèticament):

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Properties;

public class TestDB {

    public static void main(String[] args) {

        String url = "jdbc:mariadb://172.16.0.100:3306/employees";
        Properties connectionProperties = new Properties();
        connectionProperties.setProperty("user", "admin");
        connectionProperties.setProperty("password", "1234");
        try {
            Connection con = DriverManager.getConnection(url,
connectionProperties);
            Statement st = con.createStatement(); ) { ①
            System.out.println("Base de dades connectada!");
        }
    }
}
```

```

try (
    ResultSet rs = st.executeQuery("SELECT emp_no,
first_name, last_name FROM employees ORDER BY last_name, first_name
LIMIT 10");) { ❷
    System.out.println("Dades de la taula employees:");
    while (rs.next()) { ❸
        int empNo = rs.getInt("emp_no");
        String firstName = rs.getString("first_name"); ❹
        String lastName = rs.getString("last_name"); ❺
        System.out.printf("%8d %15s %15s\n", empNo,
firstName, lastName);
    }
}
}

```

- ❶ En primer lloc hem d'obtenir un objecte **Statement** a partir de la connexió. L'objecte **Statement** ens permet enviar sentències SQL a la base de dades i recuperar el resultat.
- ❷ Al resultat d'una consulta SQL s'hi pot accedir utilitzant un **ResultSet**.
- ❸ Aquest **ResultSet** és un cursor: podem passar a la següent fila amb el mètode **next()**
- ❹ i podem recuperar les dades de cadascuna de les files amb les diferents variants dels mètodes **get**, com **getInt()** o **getString()**.

Podem veure que l'obtenció de text i nombres de la base de dades no presenta cap dificultat: simplement cridem el mètode **get** adequat del **ResultSet**.

74.2. Alguns tipus de dades poden presentar dificultats

Altres tipus de dades però, com les enumeracions o les dates, poden presentar algunes dificultats addicionals:

```

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

```

```

import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.util.Properties;

public class TestDB {

    public static void main(String[] args) {
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd-
MM-yyyy");
        String url = "jdbc:mariadb://172.16.0.100:3306/employees";
        Properties connectionProperties = new Properties();
        connectionProperties.setProperty("user", "admin");
        connectionProperties.setProperty("password", "1234");
        try (Connection con = DriverManager.getConnection(url,
connectionProperties); Statement st = con.createStatement();) {
            System.out.println("Base de dades connectada!");
            try (ResultSet rs = st.executeQuery("SELECT * FROM employees
ORDER BY last_name, first_name LIMIT 10");) {
                System.out.println("Dades de la taula employees:");
                while (rs.next()) {
                    int empNo = rs.getInt("emp_no");
                    LocalDate birthDate =
rs.getDate("birth_date").toLocalDate(); ❶
                    String firstName = rs.getString("first_name");
                    String lastName = rs.getString("last_name");
                    Gender gender =
Gender.valueOf(rs.getString("gender")); ❷
                    LocalDate hireDate =
rs.getDate("hire_date").toLocalDate(); ❸
                    System.out.printf("%8d %14s %15s %15s %6s %14s\n",
empNo, birthDate.format(formatter),
firstName, lastName, gender,
hireDate.format(formatter));
                }
            }
        } catch (SQLException e) {
            System.err.println("Error SQL: " + e.getMessage());
        }
    }

    public enum Gender {
        M, F;
    }
}

```

- ❶ Recuperem la informació de data i la convertim a un tipus Java.
- ❷ Convertim el genere en un enumeració Java
- ❸ Hem de tractar el cast dels tipus data manualment.

De l'exemple anterior hi ha dos aspectes a comentar, el tractament de les dates i el tractament de les enumeracions.

La traducció entre els tipus de temps de SQL al tipus de temps en Java sempre ha estat complicada.

A partir de Java 8 es millora la correspondència entre els tipus de data i temps SQL i els de Java. Per llegir dades d'aquests tipus no cal utilitzar el mètode **getDate()** que s'utilitzava en versions anteriors, sinó que tots es poden llegir amb el mètode genèric **getObject()**.

La correspondència entre tipus SQL i tipus Java és la següent:

Taula 74.1. Correspondència entre tipus SQL i tipus Java

SQL	Java
DATE	LocalDate
TIME	LocalTime
TIMESTAMP	LocalDateTime
TIME amb fus horari	OffsetTime
TIMESTAMP amb fus horari	OffsetDateTime

El mètode **getObject()** rep el nom o posició de l'atribut a recuperar i el tipus d'objecte que volem. Per exemple:

```
LocalDate date = rs.getObject("date", LocalDate.class);
```

A l'exemple hem utilitzat un **DateTimeFormatter** per donar format a les dates a l'hora de mostrar-les per pantalla. L'hem declarat així:

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd-MM-yyyy");
```

Això indica que volem veure les dates posant primer el dia, després el mes i després l'any, i que separam cada camp amb guions.

Per utilitzar el **formatter** hem fet **birthDate.format(formatter)**, cosa que ens retorna un **String** amb les dades de la data **birthDate** expressades en el format que volem.

Per altra banda, el MySQL té un tipus de dada enumeració que no és present a altres SGBD. JDBC els recupera directament com String. Si volem mantenir el tipus enumeració, podem crear un **enum** amb els mateixos valors, i utilitzar el mètode **valueOf** de l'enum, que tradueix una cadena de text al valor corresponent de l'**enum**. Així s'ha fet a l'exemple.

74.3. Obtenir els camps d'un registre a partir de l'índex del camp

El següent exemple és similar al primer, però accedim a les columnes retornades a través del seu índex en comptes del seu nom:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Properties;

public class TestDB {

    public static void main(String[] args) {
        boolean hasResults = false;
        String url = "jdbc:mariadb://172.16.0.100:3306/employees";
        Properties connectionProperties = new Properties();
        connectionProperties.setProperty("user", "admin");
        connectionProperties.setProperty("password", "1234");
        try (Connection con = DriverManager.getConnection(url,
connectionProperties); Statement st = con.createStatement();) {
            System.out.println("Connexió a la base de dades!");
            try (ResultSet rs = st.executeQuery("SELECT first_name,
last_name FROM employees ORDER BY last_name, first_name LIMIT 10");) {
                while (rs.next()) {
                    if (!hasResults) {
                        hasResults = true;
                        System.out.println("Llista d'empleats:");
                    }
                    String firstName = rs.getString(1); ①
                }
            }
        }
    }
}
```

```

        String lastName = rs.getString(2); ②
        System.out.println(firstName + " " + lastName);
    }
    if (!hasResults) {
        System.out.println("No hi ha dades a la taula");
    }
}
} catch (SQLException e) {
    System.err.println("Error SQL: " + e.getMessage());
}
}

public enum Gender {
    M, F;
}
}

```

- ❶ El primer camp té índex 1, COMPTE!!
- ❷ Accedim al segon camp del registre.

74.4. Sel·leccions parametritzades i injecció de codi

El següent exemple mostra una situació una mica més complexa. A la BD d'empleats, cada empleat pot tenir un o més títols. Volem un programa que ens mostri tots els títols que hi ha a la base de dades, que permeti a l'usuari triar-ne un, i que finalment mostri tots els empleats que tenen el títol triat per l'usuari.

```

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

public class TestDB {

    private Connection connection = null;
    private Statement statement = null;
    private Scanner scanner = new Scanner(System.in);
}

```

```
private void connect() throws SQLException {
    String url = "jdbc:mariadb://172.16.0.100:3306/employees";
    String user = "admin";
    String passwd = "1234";
    if (connection == null) {
        connection = DriverManager.getConnection(url, user, passwd);
    }
    if (statement == null) {
        statement = connection.createStatement();
    }
}

private void disconnect() {
    if (statement != null) {
        try {
            statement.close();
        } catch (SQLException e) {
            ;
        } finally {
            statement = null;
        }
    }
    if (connection != null) {
        try {
            connection.close();
        } catch (SQLException e) {
            ;
        } finally {
            connection = null;
        }
    }
}

public List<String> getTitles() throws SQLException {
    List<String> titles = new ArrayList<String>();
    try (ResultSet rs = statement.executeQuery("SELECT DISTINCT
title FROM titles;")) {
        //Afegeix tots els títols a la llista
        while (rs.next()) {
            titles.add(rs.getString(1));
        }
    }
    return titles;
}
```

```

public String chooseTitle(List<String> titles) {
    String title = "";
    for (String t : titles) {
        System.out.println(t);
    }
    System.out.println("\nQuin títol vols? ");
    while (!titles.contains(title)) {
        title = scanner.nextLine();
        if (!titles.contains(title)) {
            System.out.println("Aquest títol no existeix.");
        }
    }
    return title;
}

public void employeesByTitle(String title) throws SQLException {
    try (ResultSet rs = statement.executeQuery("SELECT first_name,
last_name "
+ "FROM employees JOIN titles "
+ "ON employees.emp_no=titles.emp_no "
+ "WHERE title LIKE '" + title + "'")); { ①
        while (rs.next()) {
            System.out.println(rs.getString("first_name") +
rs.getString("last_name"));
        }
    }
}

public void run() {
    String title;
    try {
        //1- Connecta a la bbdd
        connect();
        //2- Mostra els títols i demana a l'usuari que en triï un
        title = chooseTitle(getTitles());
        //3- Mostra els empleats que tenen el títol seleccionat
        System.out.println("Títol seleccionat: " + title);
        employeesByTitle(title);
    } catch (SQLException e) {
        System.err.println(e.getMessage());
    } finally {
        disconnect();
    }
}

```

```
public static void main(String[] args) {  
    TestDB ex = new TestDB();  
    ex.run();  
}  
}
```

Hi ha un aspecte important a destacar en aquest exemple:

- ❶ El títol a cercar a la base de dades l'introduceix l'usuari, i l'incorporem a una sentència SQL dins del mètode **employeesByTitle**.

Per tant, qualsevol caràcter especial que l'usuari hagi introduït s'incorporarà a la sentència SQL i serà interpretat segons les regles del SQL.

Això és un problema greu de seguretat. Un usuari hàbil pot construir una expressió que faci que s'executin sentències al SGBD molt diferents a les que teníem previstes. Amb aquesta tècnica, coneguda com **SQL-Injection**, es poden obtenir dades qualsevol de la BD, i es poden modificar o eliminar registres als quals se suposa que l'usuari no ha de tenir accés.

Encara que l'usuari no ho faci intencionadament, pot incloure algun caràcter com unes cometes o similar a la seva cerca i això farà que el programa funcioni malament.

Una opció que tenim per evitar aquest problema és filtrar qualsevol entrada que vingui de fora del programa (sigui introduïda per un usuari, transmessa per una xarxa, o obtinguda d'un fitxer). En aquest filtre podríem eliminar els caràcters que tenen un significat especial en SQL.

Fer això, però, és delicat: és fàcil oblidar-se de filtrar algunes de les entrades en un programa complex, o d'equivocar-se en algun detall del filtre.

La solució passa per fer que sigui el propi JDBC qui filtri les entrades insegures. Això ho farem utilitzant una **PreparedStatement** en comptes d'un **Statement**, tal com veurem al següent apartat.

75

Sentències preparades

75.1. SQL-Injection

Observem el següent programa:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Scanner;

public class TestDB {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String lastName = " ";
        try (Connection conn = DriverManager.getConnection(
                "jdbc:mysql://172.16.0.100:3306/
employees", "admin", "1234"); Statement st = conn.createStatement();) {
            while (!lastName.equals("")) {
                System.out.println("Cognom a cercar (en blanc per
sortir): ");
                lastName = sc.nextLine();
                if (!lastName.equals("")) {
                    try (ResultSet rs = st.executeQuery(
                            "SELECT emp_no, first_name, last_name FROM
employees WHERE last_name LIKE '" + lastName + "'")); {
                        while (rs.next()) {
                            int empNo = rs.getInt("emp_no");
                            String firstName =
rs.getString("first_name");
                        }
                    }
                }
            }
        }
    }
}
```

```

        lastName = rs.getString("last_name");
        System.out.println(empNo + "\t" + firstName
+ " " + lastName);
    }
}
}
}
} catch (SQLException e) {
    System.err.println("Error SQL: " + e.getMessage());
}
sc.close();
}
}

```

Aquest programa demana a l'usuari un cognom i mostra el número i nom complet dels empleats que tenen aquest cognom:

```

Cognom a cercar (en blanc per sortir):
Flanders
14908 Kerhong Flanders
16333 Kazunori Flanders
19595 Gift Flanders
21032 Leah Flanders
24501 Constantijn Flanders
...

```

Aparentment el programa funciona correctament, però l'usuari pot introduir cadenes com:

```

Cognom a cercar (en blanc per sortir):
' or '1'='1
10001 Georgi ' or '1'='1
10002 Bezalel ' or '1'='1
10003 Parto ' or '1'='1
10004 Chirstian ' or '1'='1
10005 Kyoichi ' or '1'='1
10006 Anneke ' or '1'='1
10007 Tzvetan ' or '1'='1
10008 Saniya ' or '1'='1
10009 Sumant ' or '1'='1
10010 Duangkaew ' or '1'='1

```

L'usuari ha aconseguit mostrar la llista completa d'empleats.

Això és un problema greu de seguretat. Imaginem una consulta similar que tingui per objectiu validar l'usuari i contrasenya d'un usuari:

```
String query = "select name, country, password from Users where email =\n    " + id + "' and password='" + pwd + "'";
```

Si a aquesta consulta li envíem un usuari de l'estil `prova@domini.com' or '1='1` aconseguim validar l'accés sense necessitat de tenir una parella de nom d'usuari i contrasenya vàlids.

75.2. Ús de les sentències preparades

Per evitar aquests problemes, sempre que una sentència SQL depengui de paràmetres que poden variar o que hem obtingut de fonts no fiables, hem d'utilitzar una sentència preparada.

Les sentències preparades tenen els següents avantatges:

- Filtren els paràmetres variables de les sentències, **evitant vulnerabilitats**.
- **Simplifiquen l'escriptura de les sentències**, ja que no hem d'anar tancat i obrint cometes i concatenant cadenes.
- Si una mateixa consulta es repeteix molts cops amb paràmetres diferents, **optimitzen la velocitat** d'accés a les dades, perquè la sentència només s'envia un cop al SGBD. El SGBD compila la sentència i la guarda, modificant només els paràmetres necessaris cada vegada que s'executa la sentència.

El següent programa és la versió amb sentències preparades del programa d'exemple anterior:

```
import java.sql.Connection;\nimport java.sql.DriverManager;\nimport java.sql.PreparedStatement;\nimport java.sql.ResultSet;\nimport java.sql.SQLException;\nimport java.util.Scanner;\n\npublic class TestDB {\n\n    public static void main(String[] args) {
```

```

Scanner sc = new Scanner(System.in);
String lastName = " ";
try (Connection conn =
DriverManager.getConnection("jdbc:mysql://172.16.0.100:3306/
employees", "admin", "1234"));
    PreparedStatement st = conn.prepareStatement("SELECT
emp_no, first_name, last_name FROM employees WHERE last_name LIKE ?");)
{ ① ②
    while (!lastName.equals("")) {
        System.out.println("Cognom a cercar (en blanc per
sortir): ");
        lastName = sc.nextLine();
        if (!lastName.equals("")) {
            st.setString(1, lastName); ③
            try (ResultSet rs = st.executeQuery()); { ④
                while (rs.next()) {
                    int empNo = rs.getInt("emp_no");
                    String firstName =
rs.getString("first_name");
                    lastName = rs.getString("last_name");
                    System.out.println(empNo + "\t" + firstName
+ " " + lastName);
                }
            }
        }
    }
} catch (SQLException e) {
    System.err.println("Error SQL: " + e.getMessage());
}
sc.close();
}
}

```

Hi ha només algunes diferències:

- ① Hem creat una **PreparedStatement** amb el mètode **prepareStatement()** de la connexió.
- ② La sentència SQL a executar es passa a **prepareStatement()**, posant interrogants als llocs on volem posar els paràmetres.
- ③ Abans d'executar la sentència, utilitzem les diferents variants de **set** que té **PreparedStatement** (com **setString()**, **setInt()**, etc.) per assignar un valor concret als paràmetres. **El primer interrogant és el paràmetre número 1.**

-
- ④ Finalment, utilitzem el mètode **executeQuery()** del **PreparedStatement** per obtenir els resultats de la sentència.

- La **PreparedStatement** es pot executar múltiples vegades, modificant els seus paràmetres, sense haver de tornar a crear una consulta nova.

Si ara intentem passar la consulta problemàtica d'abans, veiem que no obtenim cap resultat:

```
Cognom a cercar (en blanc per sortir):  
' or '1'='1  
Cognom a cercar (en blanc per sortir):
```

75.3. LIKE vs =

Un detall que no té a veure amb les sentències preparades però que val la pena recordar és la diferència entre utilitzar **LIKE** o **=** per comparar cadenes.

En el nostre exemple, com que utilitzem **LIKE**, l'usuari pot introduir cadenes amb comodins, com **%ders**, obtenint tots els empleats el cognom dels quals acaba en **ders**.

Si no volem això, com en el cas de l'usuari i contrasenya, que han de ser exactes, és millor utilitzar la comparació amb **=**, que no admet comodins.

Cal tenir en compte que quan volem fer servir els comodins en una condició **LIKE**, si els posem directament dins de la cadena, (**columna LIKE "%?%"**) no ens funcionarà. En aquest cas, tant els percentatges com l'interrogant s'interpreten com a literals i, per tant, els mètodes de **PreparedStatement**, com **setString()**, ens llençaran una excepció quan no trobin els interrogants que esperaven.

Per evitar això podem generar la **PreparedStatement** sense els comodins i afegir-los posteriorment a la cadena mitjançant el mètode **setString()**.

Sentències DML i DDL. Modificació de dades

Les sentències de modificació de dades, com **INSERT**, **DELETE** o **UPDATE**, funcionen de forma similar a les sentències de consulta.

En el següent exemple inserim una nova fila a la taula **employees**:

```
public class TestDB {  
  
    public static void main(String[] args) {  
        String url = "jdbc:mysql://localhost:3306/employees";  
        Properties propietatsConnexio = new Properties();  
        propietatsConnexio.put("user", "admin");  
        propietatsConnexio.put("password", "1234");  
  
        try (Connection con = DriverManager.getConnection(url,  
propietatsConnexio); Statement st = con.createStatement();) {  
            int numFiles = st.executeUpdate("INSERT INTO  
employees(emp_no, birth_date, first_name, last_name, gender, hire_date)  
VALUES(1, '1953-03-16', 'Richard', 'Stallman', 'M', '1975-01-21')"); ①  
            System.out.println("Inserció creada! Files afectades: " +  
numFiles);  
        } catch (SQLException e) {  
            System.err.println("Error SQL: " + e.getMessage());  
        }  
    }  
}
```

-
- ❶ A diferències dels exemples anteriors, en aquest codi utilitzem **executeUpdate()** en comptes d'**executeQuery()**.

El mètode **executeUpdate()** permet enviar quasevol de les sentències de modificació de dades, així com de definició de dades (**CREATE**, **DROP**, **ALTER**); retorna un **int** que és la quantitat de files afectades pel canvi. En l'exemple anterior retornaria un 1, ja que només hem inserit una fila.

També podem utilitzar sentències preparades per a les modificacions de dades. En el següent exemple es permet a l'usuari afegir un nou idioma a la taula **language** de la base de dades **sakila**.

```
public class TestDB {

    public static void main(String[] args) {
        if (args.length == 1) {
            try (Connection connection =
DriverManager.getConnection("jdbc:mariadb://localhost/
sakila", "admin", "1234");
            PreparedStatement comptaIdioma =
connection.prepareStatement("SELECT count(*) FROM language WHERE
name=?");
            PreparedStatement insereixIdioma =
connection.prepareStatement("INSERT INTO language(name) VALUES (?)"));
            { ❶
                comptaIdioma.setString(1, args[0]);
                try (ResultSet rs = comptaIdioma.executeQuery();) {
                    if (rs.next() && rs.getInt(1) == 0) {
                        insereixIdioma.setString(1, args[0]);
                        int files = insereixIdioma.executeUpdate();
                        if (files == 1) {
                            System.out.println("S'ha inserit la fila
correctament");
                        } else {
                            System.err.println("No s'ha pogut inserir la
fila");
                        }
                    } else {
                        System.err.println("Aquest idioma ja és a la
llista");
                    }
                }
            } catch (SQLException e) {
                // Missatge retornat pel SGBD
            }
        }
    }
}
```

```
        System.err.println("Missatge: " + e.getMessage());
        // Codi d'error estàndard
        System.err.println("Estat SQL: " + e.getSQLState());
        // Codi d'error propi del fabricant del SGBD
        System.err.println("Codi de l'error: " +
e.getErrorCode());
    }
} else {
    System.err.println("S'esperava el nom d'un idioma");
}
}
```

Al final d'aquest exemple, dins del bloc **catch**, podem veure com es pot utilitzar una **SQLException** per obtenir molta informació sobre el problema que ha provocat l'excepció.

76.1. Obtenció dels *id* autogenerats

Sovint, quan inserim dades a una base de dades, la clau primària s'està generant automàticament i ens interessa tenir-la en el nostre programa per poder fer referència a la fila que acabem d'inserir.

El JDBC proporciona un mecanisme per recuperar aquests identificadors:

```
public static void insertCustomer(Customer c) throws SQLException {
try {
    PreparedStatement insCustomer = connection().prepareStatement(
        "INSERT INTO customer(store_id, first_name, last_name, email,
address_id, create_time)"
        +" VALUES (?, ?, ?, ?, ?, NOW())",
    Statement.RETURN_GENERATED_KEYS); 1
} {
    insCustomer.setInt(1, c.getStoreId());
    insCustomer.setString(2, c.getFirstName());
    insCustomer.setString(3, c.getLastName());
    insCustomer.setString(4, c.getEmail());
    insCustomer.setInt(5, c.getAddressId());
    insCustomer.executeUpdate();
    // Obtenim claus autogenerades
    ResultSet rs = insCustomer.getGeneratedKeys(); 2
    rs.next(); // Sabem que només n'hi ha una
```

```
int customerId = rs.getInt(1);
rs.close();
// Assignem l'id a l'objecte
c.setId(customerId);
}
}
```

- ① Podem passar **Statement.RETURN_GENERATED_KEYS** quan creem un **PreparedStatement**
- ② i que això ens permet utilitzar el mètode **getGeneratedKeys()** després d'haver executat la inserció per obtenir el conjunt de claus que s'han generat automàticament.



En cas d'un **Statement** podem passar **Statement.RETURN_GENERATED_KEYS** com a segon paràmetre quan cridem el mètode **execute()**.

Interfície Resultset

La interfície **Resultset** proporciona mètodes per recuperar i manipular els resultats de les consultes que retornen un conjunt de registres.

Els **Resultsets** tenen diferents característiques, el **tipus**, la **concurrència** i la possibilitat de **mantenir el cursor** després d'un **commit**.

La configuració d'un **ResultSet** es realitza en el moment de crear-lo cridant al mètode **createStatement**, **prepareStatement** o **prepareCall**.

Aquí es mostren les sobrecàrregues del mètode **createStatement** de la interfície **_Connection**:

```
createStatement(): Statement
```

```
createStatement(int resultSetType, int resultSetConcurrency): Statement
```

```
createStatement(int resultSetType, int resultSetConcurrency, int  
resultSetHoldability): Statement
```

77.1. Tipus de resultsets

El tipus d'un **ResultSet** determina el seu nivell de funcionalitat en dues àrees:

1. De quines maneres es pot manipular el cursor.
2. Com es mostren els canvis concurrents realitzats a les dades originals en l'objecte ResultSet.

TYPE_FORWARD_ONLY

- El conjunt de resultats permet únicament la navegació **cap endavant**.
- Les files del **ResultSet** mostren les dades reals de l'origen de dades i satisfan la consulta **tant en el moment en que aquesta és executada com quan les dades són retornades**.

TYPE_SCROLL_INSENSITIVE

- El conjunt de resultats permet la navegació cap **endavant** i cap **endarrere** respecta la posició actual i permet la navegació cap a una **posició absoluta**.
- Les files del **ResultSet** són "insensibles" als canvis realitzats al conjunt de dades subjacents. Contenen les files que satisfeien la consulta **en el moment de l'execució de la consulta**.

TYPE_SCROLL_SENSITIVE

- El conjunt de resultats permet la navegació cap **endavant** i cap **endarrere** respecta la posició actual i permet la navegació cap a una **posició absoluta**.
- Les files del **ResultSet** mostren les dades reals de l'origen de dades i satisfan la consulta **tant en el moment en que aquesta és executada com quan les dades són retornades**.

Per defecte els **Resultsets** són de tipus **TYPE_FORWARD_ONLY**.



No totes les bases de dades ni tots els drivers JDBC suporten tots els tipus de cursors. EL mètode **DatabaseMetaData.supportsResultSetType** retorna true

77.2. Concurrència d'un ResultSet

La concurrència d'un **ResultSet** determina les operacions d'actualització de les dades que es poden realitzar a través d'ell.

Existeixen dos tipus de nivells de concurrència:

CONCUR_READ_ONLY

Les dades mostrades per l'objecte **ResultSet** no es poden modificar a través de la seva interfície.

CONCUR_UPDATABLE

Les dades mostrades per l'objecte **ResultSet** es poden modificar a través de la seva interfície.



No totes les bases de dades ni tots els drivers JDBC suporten tots els tipus de cursors. El mètode **DatabaseMetaData.supportsResultSetConcurrency** retorna **true**

77.3. Mantenir el cursor en un ResultSet

Cridar el mètode **Connection.commit** pot tancar els objectes **ResultSet** creats durant una transacció. La propietat "holdability" determina si els cursors (els objectes **ResultSet**) es tanquen o no quan es crida a **commit**.

Les següents constants es poden proporcionar a la connexió en el moment de crear el **Statement**.

HOLD_CURSORS_OVER_COMMIT

Els objectes **ResultSet** no es tanquen automàticament quan es crida a **commit**. Aquest tipus de cursors són especialment útils si l'aplicació utilitza els **ResultSet**s com a només lectura.

CLOSE_CURSORS_AT_COMMIT

Els objectes **ResultSet** es tanquen automàticament quan es crida el mètode **commit**.

77.4. Modificar dades a través d'un ResultSet

Si el **ResultSet** té les característiques adequades es poden fer accions d'**inserció**, ***modificació i eliminació** de registres a través seu.

77.4.1. Modificar el valor d'una columna

Per modificar el valor d'una columna d'un **ResultSet** cal situar-se al registre adequat i utilitzar els mètodes **updateInt**, **updateString_**, **updateDate**, etc... indicant el nom de la columna i el nou valor.

Els mètodes anteriors només modifiquen el **ResultSet**, per modificar les dades a la base de dades caldrà cridar al final del procés el mètode **updateRow**.

Vegem-ho amb un exemple:

```
rs.absolute(5); ①
rs.updateString("name", "Laura"); ②
rs.updateRow(); ③
```

- ① Situem el cursor al cinquè registre
- ② Modifiquem el camp **name** a "Laura"
- ③ Modifiquem la fila a l'origen de dades.

77.4.2. Inserir un nou registre

Per afegir un nou registre els resultsets modificables disposen d'una fila especial que serveix d'àrea temporal on construir la nova fila que s'afegirà.

Vegem-ho amb un exemple:

```
rs.moveToInsertRow(); ①
rs.updateString("name", "Pedro"); ②
rs.updateInt("age", 35); ③
rs.updateBoolean("active", true); ④
rs.insertRow(); ⑤
rs.moveToCurrentRow(); ⑥
```

- ① Creem un nou registre d'inserció.
- ④ Afegim els camps del nou registre.
- ⑤ Inserim el registre a l'origen de dades.
- ⑥ Tornem a situar-nos a la posició del **ResultSet** on estàvem abans de la inserció.

77.4.3. Eliminar un registre existent

A través d'un **ResultSet** també és possible eliminar un registre de l'origen de dades, n'hi ha prou en cridar el mètode **deleteRow** del **ResultSet** per eliminar el registre actual.

77.5. Un exemple sencer

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Properties;
import java.util.Scanner;

public class ExercicisDB {

    private static Connection con;

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        String url = "jdbc:mysql://172.16.0.100:3306/sakila";
        Properties connectionProperties = new Properties();
        connectionProperties.setProperty("user", "admin");
        connectionProperties.setProperty("password", "1234");

        try {
            Connection con = DriverManager.getConnection(url,
connectionProperties);
            Statement st =
con.createStatement	ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATABLE); {

                try (ResultSet rs = st.executeQuery("SELECT language_id,
name FROM language ORDER BY language_id")); {
                    boolean sortir = false;
                    while (!sortir) {
                        System.out.println("[n] Mostrar registre seguent");
                        System.out.println("[p] Mostrar registre anterior");
                        System.out.println("[u] Modificar registre actual");
                        System.out.println("[i] Inserir registre");
                        System.out.println("[d] Eliminar registre");
                        System.out.println("[e] Sortir");
                        System.out.print(">> ");
                        String opcio = in.next();
                        switch (opcio.toLowerCase().trim().charAt(0)) {
                            case 'n':
                                rs.next();
                            }
                }
            }
        }
    }
}
```

```
        break;
    case 'p':
        rs.previous();
        break;
    case 'u':
        System.out.print("Nou valor: ");
        String nouValor = in.next();
        rs.updateString("name", nouValor);
        rs.updateRow();
        break;
    case 'i':
        System.out.print("Nou valor: ");
        String nouRegistre = in.next();
        rs.moveToInsertRow();
        rs.updateString("name", nouRegistre);
        rs.insertRow();
        rs.moveToCurrentRow();
        break;
    case 'd':
        rs.deleteRow();
        break;
    case 'e':
        sortir = true;
        break;
    }
    try {
        System.out.print("Registre actual -> ");
        int languageId = rs.getInt("language_id");
        String name = rs.getString("name");
        System.out.printf("%8d %15s %n", languageId,
name);
    } catch (SQLException e) {
        System.err.println("Error SQL: " +
e.getMessage());
    }

}
} catch (SQLException e) {
    System.err.println("Error SQL: " + e.getMessage());
}
}
}
```

78

Metadades

A més de recuperar les dades emmagatzemades a una base de dades, podem obtenir informació sobre la base de dades en sí: les seves taules, els tipus de les seves columnes, quines són les claus foranes, etc. Això és el que es coneix com a **metadades**.

En el següent exemple recuperarem un munt d'informació sobre la base de dades Sakila.

```
import java.sql.Connection;
import java.sql.DatabaseMetaData;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;

public class TestDB {

    public static void main(String[] args) {
        DatabaseMetaData dbmd = null;

        try (Connection connection =
DriverManager.getConnection("jdbc:mariadb://localhost/
sakila", "admin", "1234")) {
            dbmd = connection.getMetaData();
            mostraInfoBD(dbmd);
            mostraTaules(dbmd);
            mostraColumnnes(dbmd, "film");
        } catch (SQLException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

```

public static void mostraInfoBD(DatabaseMetaData dbmd) throws
SQLException {
    String nom = dbmd.getDatabaseProductName();
    String driver = dbmd.getDriverName();
    String url = dbmd.getURL();
    String usuari = dbmd.getUserName();

    System.out.println("Informació de la BD:");
    System.out.println("Nom: " + nom);
    System.out.println("Driver: " + driver);
    System.out.println("URL: " + url);
    System.out.println("Usuari: " + usuari);
}

public static void mostraTaules(DatabaseMetaData dbmd) throws
SQLException {
    try (ResultSet rs = dbmd.getTables(null, "sakila", null, null)) {
        // Obtenir només les taules:
        // String[] types = {"TABLE"};
        // rs = dbmd.getTables(null, null, null, types);
        while (rs.next()) {
            String cataleg = rs.getString(1);
            String esquema = rs.getString(2);
            String taula = rs.getString(3);
            String tipus = rs.getString(4);
            System.out.println(tipus + " - Cataleg: " + cataleg
                + ", Esquema: " + esquema + ", Nom: " + taula);
        }
    }
}

public static void mostraColumnes(DatabaseMetaData dbmd, String
taula) throws SQLException {
    System.out.println("Columnes de la taula: " + taula);
    try (ResultSet columnes = dbmd.getColumns(null, "sakila", taula,
null)) {
        while (columnes.next()) {
            String nom = columnes.getString("COLUMN_NAME"); // 4
            String tipus = columnes.getString("TYPE_NAME"); // 6
            String mida = columnes.getString("COLUMN_SIZE"); // 7
            String nula = columnes.getString("IS_NULLABLE"); // 18
            System.out.println("Columna: " + nom + ", Tipus: " +
tipus

```

```
        + ", Mida: " + mida + ", Pot ser nula? " +
nula);
    }
}
}
```

78.1. Mètode main

```
public static void main(String[] args) {
    DatabaseMetaData dbmd = null; ①

    try (Connection connection = DriverManager.getConnection
        ("jdbc:mariadb://localhost/sakila", "usuari", "contrasenya");) {
        dbmd = connection.getMetaData();
        mostraInfoBD(dbmd); ②
        mostraTaules(dbmd); ③
        mostraColumnnes(dbmd, "film"); ④
    } catch (SQLException e) {
        System.err.println(e.getMessage());
    }
}
```

- ① La classe **DatabaseMetaData** és la classe que ens dóna accés a realitzar consultes sobre les metades de la base de dades on hem connectat.
- ② Primer obtindrem informació general sobre la base de dades
- ③ Després sobre les seves taules
- ④ Finalment mostrarem informació més detallada sobre la taula **film**.

A part d'aquests exemples, podríem obtenir molta més informació que ens interessés.

78.2. Obtenir informació de la base de dades

Al següent mètode recuperem informació general com el nom de la base de dades, la URL de connexió o amb quin usuari ens hi hem connectat:

```
public static void mostraInfoBD(DatabaseMetaData dbmd) throws
SQLException {
    String nom = dbmd.getDatabaseProductName();
```

Obtenir informació de les taules de la base de dades

```
String driver = dbmd.getDriverName();
String url = dbmd.getURL();
String usuari = dbmd.getUserName();

System.out.println("Informació de la BD:");
System.out.println("Nom: "+nom);
System.out.println("Driver: "+driver);
System.out.println("URL: "+url);
System.out.println("Usuari: "+usuari);
}
```

La sortida d'aquest mètode és:

```
Informació de la BD:
Nom: MariaDB
Driver: MariaDB Connector/J
URL: jdbc:mariadb://localhost/sakila
Usuari: admin
```

78.3. Obtenir informació de les taules de la base de dades

A partir de l'objecte **DatabaseMetaData** que hem recuperat abans, podem realitzar diverses consultes. Aquestes consultes funcionen de forma similar a les consultes SQL, en el sentit en què ens retornen un **ResultSet** que podem recórrer per recuperar tots els resultats.

Al mètode següent recuperem totes les taules de la base de dades:

```
public static void mostraTaules(DatabaseMetaData dbmd) throws
SQLException {
try (ResultSet rs = dbmd.getTables(null, "sakila", null, null);) { ❶
    // Obtenir només les taules:
    // String[] types = {"TABLE"};
    // rs = dbmd.getTables(null, null, null, types);
    while (rs.next()) {
        String catalog = rs.getString(1); ❷
        String esquema = rs.getString(2); ❸
        String taula = rs.getString(3); ❹
        String tipus = rs.getString(4); ❺
        System.out.println(tipus + " - Catalòg: " + catalog +
            ", Esquema: "+esquema+", Nom: "+taula);
```

```
}
```

```
}
```

```
}
```

- ① **getTables()** rep quatre paràmetres que es poden utilitzar per indicar sobre quines taules volem recuperar informació. **null** indica que no volem restringir els resultats pel algun dels criteris.
- ② El primer paràmetre és el **cataleg**. El significat de catàleg pot variar d'un fabricant a l'altre. En el cas de MySQL el catàleg coincideix amb el nom de la base de dades.
- ③ El segon paràmetre és l'**esquema**. En MySQL, al contrari que la major part de SGBD, no fa ús dels esquemes i considera que esquema és sinònim amb el nom de la base de dades.
- ④ El tercer paràmetre es refereix al **nom de les taules** a recuperar.
- ⑤ Finalment, el quart paràmetre és un array amb els **tipus de taules** que volem recuperar. A l'exemple les estem recuperant totes, però si només voleguessim les taules pures i no les vistes, podríem indicar en aquest paràmetre un array amb el contingut {"TABLE"}.

La major part d'aquests paràmetres permet l'ús de comodins SQL (%).

El **ResultSet** que retorna **getTables()** té un total de **10 columnes**, el significat de les quals es pot consultar a l'ajuda de Java. En aquest exemple ens hem quedat només amb les més essencials.

El resultat d'executar aquest mètode és el següent:

```
BASE TABLE - Cataleg: sakila, Esquema: null, Nom: actor
BASE TABLE - Cataleg: sakila, Esquema: null, Nom: address
BASE TABLE - Cataleg: sakila, Esquema: null, Nom: category
BASE TABLE - Cataleg: sakila, Esquema: null, Nom: city
BASE TABLE - Cataleg: sakila, Esquema: null, Nom: country
BASE TABLE - Cataleg: sakila, Esquema: null, Nom: customer
BASE TABLE - Cataleg: sakila, Esquema: null, Nom: film
BASE TABLE - Cataleg: sakila, Esquema: null, Nom: film_actor
BASE TABLE - Cataleg: sakila, Esquema: null, Nom: film_category
BASE TABLE - Cataleg: sakila, Esquema: null, Nom: film_text
BASE TABLE - Cataleg: sakila, Esquema: null, Nom: inventory
BASE TABLE - Cataleg: sakila, Esquema: null, Nom: language
BASE TABLE - Cataleg: sakila, Esquema: null, Nom: payment
BASE TABLE - Cataleg: sakila, Esquema: null, Nom: rental
BASE TABLE - Cataleg: sakila, Esquema: null, Nom: staff
```

```
BASE TABLE - Cataleg: sakila, Esquema: null, Nom: store
VIEW - Cataleg: sakila, Esquema: null, Nom: actor_info
VIEW - Cataleg: sakila, Esquema: null, Nom: customer_list
VIEW - Cataleg: sakila, Esquema: null, Nom: film_list
VIEW - Cataleg: sakila, Esquema: null, Nom: nicer_but_slower_film_list
VIEW - Cataleg: sakila, Esquema: null, Nom: sales_by_film_category
VIEW - Cataleg: sakila, Esquema: null, Nom: sales_by_store
VIEW - Cataleg: sakila, Esquema: null, Nom: staff_list
```

78.4. Obtenir informació d'una taula

El mètode **mostraColumnes** ens permet obtenir informació més detallada sobre una taula en concret:

```
public static void mostraColumnes(DatabaseMetaData dbmd, String
taula) throws SQLException {
System.out.println("Columnes de la taula: "+taula);
try (ResultSet columnes = dbmd.getColumns(null, "sakila", taula,
null)) {
while (columnes.next()) {
String nom = columnes.getString("COLUMN_NAME"); // 4
String tipus = columnes.getString("TYPE_NAME"); // 6
String mida = columnes.getString("COLUMN_SIZE"); // 7
String nula = columnes.getString("IS_NULLABLE"); // 18
System.out.println("Columna: "+nom+", Tipus: "+tipus+
", Mida: "+mida+", Pot ser nul·la? "+nula);
}
}
```

Els paràmetres de **getColumns()** són similars als de **getTables()**, però l'últim paràmetre fa referència a quines columnes volem recuperar. En el nostre cas les hem recuperat totes.

El **ResultSet** que retorna **getColumns()** té un total de 24 columnes. Se n'han seleccionat algunes de significatives per a l'exemple. El resultat és el següent:

```
Columnes de la taula: film
Columna: film_id, Tipus: SMALLINT UNSIGNED, Mida: 5, Pot ser nul·la? NO
Columna: title, Tipus: VARCHAR, Mida: 255, Pot ser nul·la? NO
Columna: description, Tipus: TEXT, Mida: 65535, Pot ser nul·la? YES
Columna: release_year, Tipus: YEAR, Mida: null, Pot ser nul·la? YES
```

```
Columna: language_id, Tipus: TINYINT UNSIGNED, Mida: 3, Pot ser nul·la? NO
Columna: original_language_id, Tipus: TINYINT UNSIGNED, Mida: 3, Pot ser nul·la? YES
Columna: rental_duration, Tipus: TINYINT UNSIGNED, Mida: 3, Pot ser nul·la? NO
Columna: rental_rate, Tipus: DECIMAL, Mida: 4, Pot ser nul·la? NO
Columna: length, Tipus: SMALLINT UNSIGNED, Mida: 5, Pot ser nul·la? YES
Columna: replacement_cost, Tipus: DECIMAL, Mida: 5, Pot ser nul·la? NO
Columna: rating, Tipus: ENUM, Mida: 5, Pot ser nul·la? YES
Columna: special_features, Tipus: SET, Mida: 54, Pot ser nul·la? YES
Columna: last_update, Tipus: TIMESTAMP, Mida: 19, Pot ser nul·la? NO
```

78.5. Metadades de consultes

A banda de recuperar informació sobre la base de dades en sí, amb JDBC també podem recuperar metadades sobre els resultats de les consultes que executem.

En el següent exemple realitzem una consulta senzilla sobre la taula **film** i després examinem el format de les columnes que ha retornat la consulta. Això ho fem a través de la classe **ResultSetMetaData**:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.Statement;

public class TestDB {

    public static void main(String[] args) {
        ResultSetMetaData rsmd = null;

        try (Connection connection =
DriverManager.getConnection("jdbc:mariadb://localhost/
sakila", "admin", "1234");
            Statement st = connection.createStatement();
            ResultSet rs = st.executeQuery("SELECT * FROM film LIMIT
3")) {

            rsmd = rs.getMetaData();
            int nCols = rsmd.getColumnCount();
```

```

        System.out.println("Columnes recuperades: " + nCols);
        for (int i = 1; i <= nCols; i++) {
            System.out.println("Columna: " + i + ":");
            System.out.println(" Nom: " + rsmd.getColumnName(i));
            System.out.println(" Tipus: " +
rsmd.getColumnTypeName(i));
            System.out.println(" Pot ser nul·la? " +
(rsmd.isNullable(i) == 0 ? "No" : "Sí"));
            System.out.println(" Amplada màxima de columna: " +
rsmd getColumnDisplaySize(i));
        }
    } catch (SQLException e) {
        System.err.println(e.getMessage());
    }
}
}
}

```

El resultat d'executar aquest programa és el següent:

```

Columnes recuperades: 13
Columna: 1:
    Nom: film_id
    Tipus: SMALLINT UNSIGNED
    Pot ser nul·la? No
    Amplada màxima de columna: 5
Columna: 2:
    Nom: title
    Tipus: VARCHAR
    Pot ser nul·la? No
    Amplada màxima de columna: 255
Columna: 3:
    Nom: description
    Tipus: TEXT
    Pot ser nul·la? Sí
    Amplada màxima de columna: 65535
Columna: 4:
    Nom: release_year
    Tipus: YEAR
    Pot ser nul·la? Sí
    Amplada màxima de columna: 4
Columna: 5:
    Nom: language_id
    Tipus: TINYINT
    Pot ser nul·la? No
    Amplada màxima de columna: 3

```

```
Columna: 6:  
    Nom: original_language_id  
    Tipus: TINYINT  
    Pot ser nul·la? Sí  
    Amplada màxima de columna: 3  
Columna: 7:  
    Nom: rental_duration  
    Tipus: TINYINT  
    Pot ser nul·la? No  
    Amplada màxima de columna: 3  
Columna: 8:  
    Nom: rental_rate  
    Tipus: DECIMAL  
    Pot ser nul·la? No  
    Amplada màxima de columna: 6  
Columna: 9:  
    Nom: length  
    Tipus: SMALLINT UNSIGNED  
    Pot ser nul·la? Sí  
    Amplada màxima de columna: 5  
Columna: 10:  
    Nom: replacement_cost  
    Tipus: DECIMAL  
    Pot ser nul·la? No  
    Amplada màxima de columna: 7  
Columna: 11:  
    Nom: rating  
    Tipus: CHAR  
    Pot ser nul·la? Sí  
    Amplada màxima de columna: 5  
Columna: 12:  
    Nom: special_features  
    Tipus: CHAR  
    Pot ser nul·la? Sí  
    Amplada màxima de columna: 54  
Columna: 13:  
    Nom: last_update  
    Tipus: TIMESTAMP  
    Pot ser nul·la? No  
    Amplada màxima de columna: 19
```

78.6. Sentències explain

Les sentències **explain** ens permeten analitzar el rendiment de les nostres consultes:

```
MariaDB [sakila]> explain select * from film order by length limit 3;
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
| id   | select_type | table | type  | possible_keys | key    | key_len
| ref  | rows       | Extra  |        |              |        |          |
+-----+-----+-----+-----+-----+-----+
|   1  | SIMPLE      | film   | ALL   | NULL          |        | NULL   |
| NULL | 1000        | Using filesort |        |              |        |          |
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

Aquí podem veure que recuperar les tres pel·lícules de menor durada implica consultar 1000 files de la taula **film** (totes les que hi ha).

```
MariaDB [sakila]> explain select * from film order by film_id limit 3;
+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+
| id   | select_type | table | type  | possible_keys | key    | key_len
| ref  | rows       | Extra  |        |              |        |          |
+-----+-----+-----+-----+-----+-----+
|   1  | SIMPLE      | film   | index | NULL          | PRIMARY | 2
| NULL | 3           |        |        |              |        |          |
+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

MariaDB [sakila]> explain select * from film order by language_id
limit 3;
+-----+-----+-----+-----+
+-----+-----+-----+-----+
| id   | select_type | table | type  | possible_keys | key    |
| key_len | ref   | rows  | Extra  |
+-----+-----+-----+-----+
+-----+-----+-----+-----+
|   1  | SIMPLE      | film   | index | NULL          |
| idx_fk_language_id | 1       | NULL  | 3   |
+-----+-----+-----+-----+
+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

En canvi, aquestes dues últimes consultes permeten la utilització d'un índex per tal de localitzar les pel·lícules a retornar sense haver de recórrer-les totes. Això permet que per retornar tres pel·lícules n'hi hagi prou amb consultar tres pel·lícules.

En la primera consulta s'utilitza la clau primària de **film** com a índex de l'ordenació, mentre que en la segona s'utilitza la clau forània de **language**.

Podem realitzar aquestes consultes des de Java de la següent manera:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class TestDB {

    public static void main(String[] args) {
        try (Connection connection =
DriverManager.getConnection("jdbc:mariadb://localhost/
sakila", "admin", "1234");
            Statement st = connection.createStatement();
            ResultSet rs = st.executeQuery("explain select * from
film order by language_id limit 3")) {
            while (rs.next()) {
                System.out.println("Clau utilitzada: " +
rs.getString("key"));
                System.out.println("Files consultades: " +
rs.getInt("rows"));
            }
        } catch (SQLException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

79

Procediments emmagatzemats

Els SGBD permeten guardar com a part de les dades d'una BD procediments i funcions que treballin sobre la resta de dades de la BD.

La utilització de procediments emmagatzemats per a realitzar les operacions més comunes contra la base de dades permet reaprofitar aquests procediments en diverses aplicacions i diferents llenguatges de programació que accedeixin a la base de dades.

També fan que sigui més fàcil de modificar les consultes en cas que canviï l'esquema de la base de dades i permeten l'optimització d'algunes consultes.

En aquest apartat veurem com podem cridar des de Java a funcions i procediments emmagatzemats.

Com que cada sistema utilitza un llenguatge i formats diferents per als seus procediments emmagatzemats, en Java s'utilitza un conveni unificat, que després es tradueix al format de crida específic del connector que estem utilitzant. D'aquesta manera podem cridar procediments de qualsevol SGBD de la mateixa manera.

79.1. Configuració de MariaDB

Abans de provar la crida de procediments des de Java, anem a veure com podem saber quins procediments hi ha creats en una BD i com ho hem de fer per cridar-los nativament.

Per començar, l'usuari que utilitzem per fer la connexió ha de tenir permisos per executar aquests procediments. En MySQL/MariaDB això es fa donant el

permís **select** sobre la taula **proc** de la BD **mysql**, que és on es guarden aquests procediments i funcions.

La sentència que otorga aquest permís és:

```
grant select on mysql.proc to <usuari>;
```

I podem veure els permisos actuals d'un usuari amb:

```
MariaDB [(none)]> show grants for admin;
+-----+
| Grants for admin@%
+-----+
| GRANT ALL PRIVILEGES ON *.* TO "admin"@"%" IDENTIFIED BY
  PASSWORD '*A4B6157319038724E3560894F7F932C8886EBFCF' |
+-----+
| 1 row in set (0.000 sec)
```

Per veure quins procediments i funcions hi ha a la base de dades actual podem utilitzar:

```
MariaDB [sakila]> show procedure status;
+-----+-----+-----+-----+
| Db    | Name          | Type   | Definer      | Modified
|       | Created        |         | Security_type | Comment
|       |                | character_set_client | collation_connection |
| Database Collation |
```

Db	Name	Type	Definer	Modified
	Created	Security_type	Comment	character_set_client collation_connection
sakila	film_in_stock	PROCEDURE	root@localhost	2016-01-20 17:30:10 2016-01-20 17:30:10 DEFINER utf8
				utf8_general_ci latin1_swedish_ci

```
| sakila | film_not_in_stock | PROCEDURE | root@localhost
| 2016-01-20 17:30:10 | 2016-01-20 17:30:10 | DEFINER           |
|                               | utf8               |
utf8_general_ci      | latin1_swedish_ci  |
| sakila | rewards_report    | PROCEDURE | root@localhost
| 2016-01-20 17:30:10 | 2016-01-20 17:30:10 | DEFINER           |
Provides a customizable report on best customers | utf8
| utf8_general_ci      | latin1_swedish_ci  |
+-----+-----+-----+
+-----+-----+-----+
+-----+-----+-----+
+-----+-----+-----+
3 rows in set (0.00 sec)

MariaDB [sakila]> show function status;
+-----+-----+-----+
+-----+-----+-----+
+-----+-----+-----+
| Db      | Name          | Type   | Definer   |
| Modified | Created       | Security_type | Comment |
| character_set_client | collation_connection | Database Collation |
+-----+-----+-----+
+-----+-----+-----+
+-----+-----+-----+
| sakila | get_customer_balance | FUNCTION | root@localhost
| 2016-01-20 17:30:10 | 2016-01-20 17:30:10 | DEFINER           |
| utf8      | utf8_general_ci | latin1_swedish_ci |
| sakila | inventory_held_by_customer | FUNCTION | root@localhost
| 2016-01-20 17:30:10 | 2016-01-20 17:30:10 | DEFINER           |
| utf8      | utf8_general_ci | latin1_swedish_ci |
| sakila | inventory_in_stock | FUNCTION | root@localhost
| 2016-01-20 17:30:10 | 2016-01-20 17:30:10 | DEFINER           |
| utf8      | utf8_general_ci | latin1_swedish_ci |
+-----+-----+-----+
+-----+-----+-----+
+-----+-----+-----+
3 rows in set (0.00 sec)
```

79.2. Crida a procediments

També podem veure el codi font de qualsevol d'aquestes funcions i procediments:

```
MariaDB [sakila]> show create procedure film_in_stock;
```

```
+-----+
+-----+
+-----+
| Procedure      | sql_mode
+-----+
| Create Procedure
+-----+
| character_set_client | collation_connection |
Database Collation |
+-----+
+-----+
+-----+
| film_in_stock |
| STRICT_TRANS_TABLES,STRICT_ALL_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION |
| CREATE DEFINER=`root`@`localhost` PROCEDURE `film_in_stock`(IN p_film_id INT, IN p_store_id INT, OUT p_film_count INT)
|           READS SQL DATA
| BEGIN
|       SELECT inventory_id
|         FROM inventory
|        WHERE film_id = p_film_id
|          AND store_id = p_store_id
|          AND inventory_in_stock(inventory_id);
|
|       SELECT FOUND_ROWS() INTO p_film_count;
| END | utf8               | utf8_general_ci      | latin1_swedish_ci  |
+-----+
+-----+
+-----+
| 1 row in set (0.00 sec)
```

En aquest exemple, podem veure que el procediment **film_in_stock** rep tres paràmetres, tots de tipus enter. Els dos primers són d'entrada (**IN**), és a dir, s'utilitzen per passar dades cap al procediment. El tercer, en canvi, és de sortida (**OUT**), així que aquí rebrem algun tipus de resposta després d'haver cridat el procediment.

Podem veure que el codi del procediment realitza una consulta SQL i, a l'última línia, es guarda la quantitat de files retornades a aquest tercer paràmetre.

Cridem el mètode per veure els exemplars de la pel·lícula amb id 4 que hi ha a la tenda amb id 1:

```
MariaDB [sakila]> call film_in_stock(3,1,@n);
Empty set (0.00 sec)

Query OK, 1 row affected (0.00 sec)

MariaDB [sakila]> call film_in_stock(4,1,@n);
+-----+
| inventory_id |
+-----+
|      16 |
|      17 |
|      18 |
|      19 |
+-----+
4 rows in set (0.02 sec)

Query OK, 1 row affected (0.02 sec)

MariaDB [sakila]> select @n;
+---+
| @n |
+---+
|   4 |
+---+
1 row in set (0.00 sec)
```

Podem veure que hi ha disponibles els ítems 16, 17, 18 i 19. A la variable **n** s'ha guardat el total d'ítems disponibles, 4.

79.3. Crida a funcions

Fem el mateix ara amb una funció:

```
MariaDB [sakila]> show create function get_customer_balance;
```

```
+-----+-----+-----+
| Function          | sql_mode
| Create Function
| character_set_client
| collation_connection | Database Collation |
+-----+
+
+
+-----+-----+-----+
| get_customer_balance |
STRICT_TRANS_TABLES, STRICT_ALL_TABLES, NO_ZERO_IN_DATE, NO_ZERO_DATE, ERROR_FOR_DIVISION_BY_ZERO, NO_AUTO_CREATE_USER, NO_AUTO_VALUE_ON_ZERO
| CREATE DEFINER=`root`@`localhost` FUNCTION
`get_customer_balance`(p_customer_id INT, p_effective_date
DATETIME) RETURNS decimal(5,2)
READS SQL DATA
DETERMINISTIC
BEGIN
    DECLARE v_rentfees DECIMAL(5,2);
    DECLARE v_overfees INTEGER;
    DECLARE v_payments DECIMAL(5,2);

    SELECT IFNULL(SUM(film.rental_rate),0) INTO v_rentfees
```

```

FROM film, inventory, rental
WHERE film.film_id = inventory.film_id
    AND inventory.inventory_id = rental.inventory_id
    AND rental.rental_date <= p_effective_date
    AND rental.customer_id = p_customer_id;

SELECT IFNULL(SUM(IF((TO_DAYS(rental.return_date) -
TO_DAYS(rental.rental_date)) > film.rental_duration,
((TO_DAYS(rental.return_date) - TO_DAYS(rental.rental_date)) -
film.rental_duration),0)),0) INTO v_overfees
FROM rental, inventory, film
WHERE film.film_id = inventory.film_id
    AND inventory.inventory_id = rental.inventory_id
    AND rental.rental_date <= p_effective_date
    AND rental.customer_id = p_customer_id;

SELECT IFNULL(SUM(payment.amount),0) INTO v_payments
FROM payment

WHERE payment.payment_date <= p_effective_date
AND payment.customer_id = p_customer_id;

RETURN v_rentfees + v_overfees - v_payments;
END | utf8           | utf8_general_ci      | latin1_swedish_ci  |
+-----+
+-----+
+-----+
+-----+
+-----+
1 row in set (0.00 sec)

```

Les funcions d'usuari es poden cridar com qualsevol altra funció de MySQL.

Per exemple, podem esbrinar quins són els usuaris que ens deuen diners:

```

MariaDB [sakila]> select customer_id, get_customer_balance(customer_id,
now()) as balance from customer having balance!=0;
+-----+-----+
| customer_id | balance |
+-----+-----+
|      546 |   -3.99 |
|      554 |   -3.00 |
|       16 |   -1.99 |
|     259 |   -1.99 |

```

```
|      401 |    -0.99 |
|      577 |    -0.99 |
+-----+-----+
6 rows in set (0.45 sec)
```

79.4. Crida a procediments des de Java

Per fer la mateixa crida que hem fet abans podríem utilitzar el següent codi:

```
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Types;

public class TestDB {

    public static void main(String[] args) {
        String sql = "{call film_in_stock(4,1,?)}";

        try (Connection connection =
DriverManager.getConnection("jdbc:mariadb://localhost/
sakila", "admin", "1234");
        CallableStatement st = connection.prepareCall(sql)) { ❶

            st.registerOutParameter(1, Types.INTEGER); ❷

            ResultSet rs = st.executeQuery(); ❸
            while (rs.next()) {
                System.out.println(rs.getInt(1));
            }
            System.out.println("Files retornades: " + st.getInt(1));
        } catch (SQLException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

- ❶ Hem utilitzat el mètode **prepareCall()** de la connexió per preparar la crida, de la mateixa manera que ho fem amb els **PreparedStatement**. En aquest cas, el mètode ens retorna un **CallableStatement**.

- ② Després hem omplert el paràmetre que hi mancava, indicant que és un paràmetre de sortida de tipus enter.
- ③ Finalment, l'execució del procediment ens retorna un **ResultSet** amb totes les files obtingudes, i podem recuperar el paràmetre de sortida amb **getInt()**. També podem generalitzar la consulta, i posar interrogants també pels paràmetres d'entrada:

```
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Types;

public class TestDB {

    public static void main(String[] args) {
        String sql = "{call film_in_stock(?, ?, ?)}";

        try (Connection connection =
DriverManager.getConnection("jdbc:mariadb://localhost/
sakila", "admin", "1234");
        CallableStatement st = connection.prepareCall(sql)) {

            st.setInt(1, 4);
            st.setInt(2, 1);

            st.registerOutParameter(1, Types.INTEGER);

            ResultSet rs = st.executeQuery();
            while (rs.next()) {
                System.out.println(rs.getInt(1));
            }
            System.out.println("Files retornades: " + st.getInt(1));
        } catch (SQLException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

79.5. Crides a funcions des de Java

La crida de funcions d'usuari no mereix especial atenció, ja que les funcions es criden de la mateixa manera que qualsevol altra funció de MySQL:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class TestDB {

    public static void main(String[] args) {
        String sql = "select get_customer_balance(?, now()) as balance";

        try (Connection connection =
DriverManager.getConnection("jdbc:mariadb://localhost/
sakila", "admin", "1234");
        PreparedStatement st =
connection.prepareStatement(sql)) {
            st.setInt(1, 259);

            ResultSet rs = st.executeQuery();
            while (rs.next()) {
                System.out.println(rs.getDouble(1));
            }
        } catch (SQLException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

80

Transaccions

Sovint ens trobem en situacions en què hem de fer diverses modificacions a les dades d'una BD i garantir que totes aquestes modificacions es duen a terme, o ve que cap ho fa.

Per exemple, en un traspàs de diners d'un compte bancari a un altre s'han de fer dues operacions: retirar uns diners del compte origen i ingressar-los al compte de destí. No hauríem de permetre que per una errada en algun punt (el servidor de BD, la xarxa, el nostre programa...), és fes una de les operacions però l'altra no.

Les **transaccions** permeten agrupar diverses operacions contra una base de dades i tractar-les com si es tractés d'una sola operació. En cas que el procés falli, tenim l'opció de desfer aquelles operacions que ja s'hagin enviat. Un cop realitzades totes les operacions de la transacció correctament podem confirmar-la, de manera que tots els canvis passen definitivament a la base de dades.

80.1. Autocommit

Si no diem el contrari, quan treballem amb JDBC contra MySQL tenim la funcionalitat d'**autocommit** activada. Això significa que cadascuna de les operacions que realitzem contra la base de dades es considera una transacció que és confirmada automàticament quan s'ha executat amb èxit.

L'avantatge d'això és que ens podem oblidar de les transaccions i sabem que qualsevol cosa que puguem expressar com una única sentència SQL s'executarà sincera o fallarà sense fer canvis a la base de dades.

El problema però ve quan necessitem que diverses sentències s'executin com una única transacció. Aleshores hem de deshabilitar l'**autocommit** i agafar

nosaltres el control per indicar quan comença i acaba cadascuna de les transaccions.

Amb l'**autocommit** desactivat, la totalitat de consultes executades no són definitives fins que es crida el mètode **commit()** explícitament.

En canvi, si volem anul·lar tots els canvis que s'han fet dins de la transacció, ho podem fer amb el mètode **rollback()**.

80.2. Exemple

Al següent exemple apujarem el sou a un empleat. L'empleat en qüestió és en Georgi Facello i volem apujar-li el sou un 5% sobre el seu sou actual. Cal vigilar perquè hi ha dos empleats amb el mateix nom. En el nostre cas volem augmentar el sou de l'empleat contractat el 26 de juny de 1986.

Modificar un sou implica dues operacions sobre la taula **salaries**:

1. posar com a data de finalització del sou actual el dia d'avui,
2. i afegir-hi una fila nova fent constar el nou sou, la data d'inici i, com a data final, posar el valor especial 9999-01-01.

Per garantir la coherència de les dades hem d'assegurar que aquestes dues operacions es fan dins de la mateixa transacció:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class ExercicisDB {

    public static void main(String[] args) {
        int empNo = 0;
        int currentSalary = 0;

        // Sentències SQL que necessitarem
        String findEmpNoSQL = "select emp_no from employees where
first_name=? and last_name=? and hire_date=?";
        String findCurrentSalarySQL = "select salary from salaries where
emp_no=? and to_date='9999-01-01'";
```

```

String closeSalarySQL = "update salaries set to_date=curdate()
where emp_no=? and to_date='9999-01-01'";
String openSalarySQL = "insert into salaries values (?, ?, 
curdate(), '9999-01-01')";

// Establim connexió
try (Connection con =
DriverManager.getConnection("jdbc:mariadb://172.16.0.100:3306/
employees", "admin", "1234")) {
    // Creem els PreparedStatement
    try (PreparedStatement findEmpNoSt =
con.prepareStatement(findEmpNoSQL);
        PreparedStatement findCurrentSalarySt =
con.prepareStatement(findCurrentSalarySQL);
        PreparedStatement closeSalarySt =
con.prepareStatement(closeSalarySQL);
        PreparedStatement openSalarySt =
con.prepareStatement(openSalarySQL)) {
        // Desactivem autocommit per iniciar transacció
        con.setAutoCommit(false); ①

        // Cerquem el número de l'empleat
        findEmpNoSt.setString(1, "Georgi");
        findEmpNoSt.setString(2, "Facello");
        findEmpNoSt.setString(3, "1986-06-26");
        try (ResultSet rs = findEmpNoSt.executeQuery()) {
            if (rs.next()) {
                empNo = rs.getInt(1);
                if (rs.next()) {
                    throw new SQLException("Hi ha més d'un
treballador amb aquestes dades.");
                }
            } else {
                throw new SQLException("No hi ha cap treballador
amb aquestes dades.");
            }
        }

        // Cerquem el sou actual de l'empleat
        findCurrentSalarySt.setInt(1, empNo);
        try (ResultSet rs = findCurrentSalarySt.executeQuery())
{
            rs.next();
            currentSalary = rs.getInt(1);
}
    }
}

```

```
// Finalitzem el sou actual
closeSalarySt.setInt(1, empNo); ②
closeSalarySt.executeUpdate();
// Assignem el nou sou
openSalarySt.setInt(1, empNo);
openSalarySt.setInt(2, (int) Math.round(currentSalary
* 1.05)); ③
openSalarySt.executeUpdate();

// Confirmem els canvis per finalitzar la transacció
con.commit(); ④
System.out.println("Transacció realitzada!");
} catch (SQLException e) {
    // En cas d'error, tornem la base de dades al seu estat
    inicial
    System.err.print("S'ha produït una errada a la
transacció, desfem els canvis");
    con.rollback(); ⑤
} finally {
    // Finalment reactivem l'autocommit
    con.setAutoCommit(true); ⑥
}
} catch (SQLException e) {
    System.err.println("Error establint connexió: " +
e.getMessage());
}
}
```

Els passos a seguir per fer una transacció sempre són els mateixos:

- ① Posar **autocommit** a **false**.
- ② Realitzar totes les actualitzacions de dades.
- ③ Si no hi ha hagut error i els resultats han estat els esperats, confirmar els canvis amb **commit**.
- ④ Si hi ha hagut algun error, desfer els canvis amb **rollback**.
- ⑤ Tornar a posar l'**autocommit** a **true** per a no afectar a la resta del programa.

Bibliografia

Llibres

Kishori Sharan. Beginning Java 8 APIs, Extensions and Libraries: Swing, JavaFX, JavaScript, JDBC and Network Programming APIs (Expert's Voice in Java). 2014. Apress. ISBN 978-1-4302-6661-7

Web

Joan Queralt. DAM-2n-POO-i-acces-a-dades. <https://gitlab.com/joanq/DAM-2n-POO-i-acces-a-dades>

Oscar Blancarte. Data Access Object (DAO) Pattern. <https://www.oscarblancarteblog.com/2018/12/10/data-access-object-dao-pattern/>

Apèndix A. Manual d'estil

Sumari

A.1. Avís important	908
A.2. Variables	909
A.2.1. Notació "Camel-case"	909
A.2.2. Constants	909
A.2.3. Números màgics	910
A.2.4. Utilitzar noms que mostrin les intencions	910
A.2.5. Evitar caràcters no americans als noms de les variables	911
A.3. Format	911
A.3.1. Sagnat	911
A.3.2. Format vertical	913
A.4. Control de flux	915
A.4.1. Bucles for	915
A.4.2.ús de break i continue	916
A.4.3.ús de System.exit()	916
A.5. Funcions i mètodes	916
A.5.1. Han de tenir un mida reduïda	916
A.5.2. Han de fer una sola cosa	916
A.5.3. Han de tenir noms descriptius	917
A.5.4. Han de tenir pocs paràmetres d'entrada	917
A.5.5. El nom dels paràmetres ha de ser explícit	917
A.6. Comentaris	917
A.6.1. Els comentaris no compensen el codi incorrecte	917
A.6.2. És millor que el codi sigui autoexplicatiu	918
A.6.3. Bons comentaris	918
A.6.4. Mals comentaris	918
A.7. Classes	919
A.7.1. Notació	919
A.7.2. Noms de classes	919
A.8. Excepcions	920
A.8.1. Excepcions genèriques	920
A.8.2. No amagar les excepcions	920

A.1. Avís important



No seguir el manual d'estil comportarà una baixada de la nota tant de les pràctiques com dels exàmens fins al punt de considerar-les no entregades i per tant avaluades amb un zero.

A.2. Variables

A.2.1. Notació "Camel-case"

Totes les variables, **no constants**, en Java s'escriuen en notació **camel-case** això és:

- **Sempre** comencen en minúscula.
- Si el nom de la variable és un nom compost, la primera lletra de cada paraula, excepte la primera, s'escriu en majúscula.
- Les demés lletres s'escriuen totes en minúscula.
- No s'utilitza el guió baix per separar les diferents parts del nom d'una variable.

Correcte.

```
metodePagament  
comptador  
diaDelMes  
fitxerSortida
```

Incorrecte.

```
MetodePagament  
COMPTADOR  
diadelmes  
fitxer_sortida
```



Aquestes convencions fan referència al llenguatge Java, altres llenguatges utilitzen diferents convencions que poden no tenir res a veure amb aquestes.

A.2.2. Constants

Si la variable es tracta com una constant s'escriu en majúscula i si es tracta de noms compostos es separa cada part del nom amb un guió baix.

Correcte.

```
MAX_CONNEXIONS  
PI
```

A.2.3. Números màgics

No utilitzar números màgics. Un número màtic és una constant numèrica dins del codi font.

Fins i tot la constant còsmica més raonable canviarà algun dia. Penseu que un any té 365 dies? Els vostres compradors marcians no estaran massa contents amb els vostres prejudicis.

Incorrecte.

```
if (p.getX() < 300)
```

Correcte.

```
final double AMPLADA_FINESTRA = 300;
...
if (p.getX() < AMPLADA_FINESTRA)
```

A.2.4. Utilitzar noms que mostrin les intencions

Si un nom de variable necessita un comentari, vol dir que no indica la seva funció.

Incorrecte.

```
int d; // Temps en transcorregut en dies
int contador; // Conta el nombre de paraules de la cadena
```

Correcte.

```
int diesDesDeLaCreacio;
int nombreDeParaules;
```



Algunes variables sense significat semàntic no necessiten cap comentari perquè queda clar quin és el seu paper, en aquest cas es poden fer servir.

Per exemple:

```
for (int i=0, i<10;i++) {
    ...
}
```



També és útil tenir en compte que els noms de les variables:

- Es puguin pronunciar.
- Es puguin buscar.

A.2.5. Evitar caràcters no americans als noms de les variables

No sabem ni la codificació del fitxer amb el codi font ni la regió del món on es pot examinar.

Dins el codi no hauria d'aparèixer cap caràcter fora del primer bloc (127 primers caràcters) de la codificació ASCII.

Això vol dir que no han d'aparèixer, ni accents, ni la c trencada ni símbols potencialment perillosos com el símbol del euro.

La única excepció a això és a dins d'un **comentari** o a dins d'una **cadena**.

A.3. Format

El format del codi és important!!

El format del codi font té a veure amb la comunicació, i la comunicació és la principal habilitat en un projecte de desenvolupament de software.

El "fer que funcioni" és secundari si el codi font no s'entén o no és fàcilment llegible.

Hi ha moltes possibilitats que la funcionalitat que creeu avui canviï en la pròxima versió, però la llegibilitat del codi tindrà un efecte profund en tots els canvis que es realitzaran d'ara fins al final de la vida del producte. El vostre estil i disciplina sobreuirà encara que el vostre codi font no ho faci.

A.3.1. Sagnat

- El contingut d'un bloc de codi sempre es sagna cap a la dreta respecte la línia que obre el bloc.
- El codi es torna a sagnar cap a l'esquerra a la línia en que el bloc finalitza.

- El sagnat cap a la dreta i cap a l'esquerra ha de ser exactament de la mateixa mida.
- La línia que obre un bloc i la línia que el tanca han d'estar alineades horitzontalment.
- Aquest sagnat pot ser de dos, tres o quatre espais però cal ser consistent, tot el codi ha de mantenir el mateix sagnat.



Recordeu que no cal posar claus en un bloc d'una sola línia.



És preferible fer servir espais que tabulacions, la codificació del codi tabulador pot donar problemes en alguns editors.

La majoria d'editors de text, però, permeten configurar que la tecla de tabulació no imprimeixi el caràcter de tabulació sinó un nombre determinat d'espais.

Incorrecte.

```
for(int i=0; i<10; i++)
System.out.println(i);
```

Correcte.

```
for(int i=0; i<10; i++)
    System.out.println(i);
```

Incorrecte.

```
while (a < 10) {
if(a % 2 == 0) {
System.out.println(a + " és parell");
}
}
```

Incorrecte.

```
while (a < 10) {
    if(a % 2 == 0) {
        System.out.println(a + " és parell");
    }
}
```

```
}
```

Correcte.

```
while (a < 10) {
    if(a % 2 == 0) {
        System.out.println(a + " és parell");
    }
}
```

A.3.2. Format vertical

El codi ha de ser fàcil de llegir verticalment, de dalt cap a baix.

- Separar els grups de línies conceptualment relacionades amb una línia en blanc.

Incorrecte.

```
package fitnesse.wikitext.widgets;
import java.util.regex.*;
public class BoldWidget extends ParentWidget {
    public static final String REGEXP = "'''.+?''''";
    private static final Pattern pattern = Pattern.compile("'''('.+?)'''", Pattern.MULTILINE + Pattern.DOTALL);
    public BoldWidget(ParentWidget parent, String text) throws
Exception {
        super(parent);
        Matcher match = pattern.matcher(text);
        match.find();
        addChildWidgets(match.group(1));
    }
    public String render() throws Exception {
        StringBuffer html = new StringBuffer("<b>");
        html.append(childHtml()).append("</b>");
        return html.toString();
    }
}
```

Correcte.

```
package fitnesse.wikitext.widgets;
```

```

import java.util.regex.*;

public class BoldWidget extends ParentWidget {
    public static final String REGEXP = "^(.+?)";
    private static final Pattern pattern = Pattern.compile("^(.+?)",
+?)"", Pattern.MULTILINE + Pattern.DOTALL);

    public BoldWidget(ParentWidget parent, String text) throws
Exception {
        super(parent);
        Matcher match = pattern.matcher(text);
        match.find();
        addChildWidgets(match.group(1));
    }

    public String render() throws Exception {
        StringBuffer html = new StringBuffer("<b>");
        html.append(childHtml()).append("</b>");
        return html.toString();
    }
}

```

- No separar els grups de línies conceptualment relacionades amb línies en blanc.

Incorrecte.

```

public class ReporterConfig {

    /**
     * The class name of the reporter listener
     */
    private String m className;

    /**
     * The properties of the reporter listener
     */
    private List<Property> m properties = new ArrayList<Property>();

    public void addProperty(Property property) {
        m properties.add(property);
    }
}

```

Correcte.

```
public class ReporterConfig {

    private String m_className;
    private List<Property> m_properties = new ArrayList<Property>();

    public void addProperty(Property property) {
        m_properties.add(property);
    }
}
```

A.4. Control de flux

A.4.1. Bucles for

Utilitzar un bucle for només en el cas que una variable va d'un valor a un altre valor amb un **increment constant**.

Correcte.

```
for (int i = 0; i < a.length; i++) {
    System.out.println(a[i]);
}
```

No utilitzar un bucle **for** per construccions estranyes com ara:

Incorrecte.

```
for (a = a / 2; count < ITERATIONS; System.out.println(xnew));
```

Un bucle com l'anterior és un bucle while.

Correcte.

```
a = a / 2;
while (count < ITERATIONS) // OK
{
    ...
    System.out.println(xnew);
}
```

A.4.2. Ús de *break* i *continue*

Evitar al **màxim** l'ús de les instruccions *break* i *continue*.

Millor utilitzar variables de control per gestionar el flux d'execució.



Si que està permès utilitzar la sentència *break* en una estructura *switch*.

A.4.3. Ús de *System.exit()*

No utilitzar *System.exit()* per finalitzar el flux d'execució d'un programa.

A.5. Funcions i mètodes

A.5.1. Han de tenir un mida reduïda

Unes 20 línies màxim.

Això implica que les instruccions *if*, *else*, *while* i similars probablement tindran una línia de longitud, una crida a una funció.

A.5.2. Han de fer una sola cosa

Les funcions només han de fer una cosa, fer-la bé i ha de ser la única cosa que facin.

Incorrecte.

```
int[] invertirVector(int[] vector) {
    int[] vectorInvertit = new int[vector.length];
    for(int i = 0; i < vector.length; i++) {
        vectorInvertit[i] = vector[vector.length - 1 - i];
        System.out.print(vectorInvertit[i] + " ");
    }
    return vectorInvertit;
}
```

Correcte.

```
int[] invertirVector(int[] vector) {
```

```

int[] vectorInvertit = new int[vector.length];
for(int i = 0; i < vector.length; i++) {
    vectorInvertit[i] = vector[vector.length - 1 - i]
}
return vectorInvertit;
}

void mostrarVector(int[] vector) {
    for(int i = 0; i < vector.length; i++) {
        System.out.print(vector[i] + " ");
    }
}

```

A.5.3. Han de tenir noms descriptius

El nom d'una funció ha de descriure perfectament la seva missió. Recordeu que una funció ha de realitzar una sola cosa, per tant el nom de la funció ha de ser justament la cosa que implementa la funció.

A.5.4. Han de tenir pocs paràmetres d'entrada

EL nombre ideal d'arguments d'una funció és zero, després un i finalment dos, evitar les funcions amb més de dos arguments.

A.5.5. El nom dels paràmetres ha de ser explícit

El nom dels paràmetres ha de ser explícit, especialment si són **integer** o **boolean**.

Incorrecte.

```
public void afegirAlumne(String s1, String s2, String s3)
```

Correcte.

```
public void afegirAlumne(String nom, String cognoms, String dni)
```

A.6. Comentaris

A.6.1. Els comentaris no compensen el codi incorrecte

Una de les motivacions més comuns per escriure comentaris és el codi mal escrit.

Escrivim un mòdul de codi confús i desorganitzat, **sabem que és confús i desorganitzat** i per això decidim comentar-lo.

El que hem de fer és reescriure'l de manera que deixi de ser confús i desorganitzat.

A.6.2. És millor que el codi sigui autoexplicatiu

Un codi autoexplicatiu evita la necessitat de posar comentaris:

```
// Comprovem si el treballador és candidat a una bonificació
if ((treballador.edat > 65) && (treballador.horesSetmanalsTreballades
    >= 40))
```

És millor:

```
if(treballadors.esCandidatABonificacio())
```

A.6.3. Bons comentaris

- Comentaris legals
- Comentaris informatius
 - Resumeixen la intenció d'un mòdul o classe de manera que permeten entendre el seu objectiu amb un cop d'ull.
- Avisar de les conseqüències de canviar o executar un determinat bloc de codi.
- Ressaltar la importància d'alguna cosa que pot semblar insubstancial.
- Documentar les APIs públiques.

A.6.4. Mals comentaris

- Escriure un comentari per que toca sense invertir el temps necessari en assegurar-se que és el millor comentari que es pot escriure.
- Descriure literalment el que està fent el codi.

Incorrecte.

```
// Mètode d'utilitat que retorna quan this.closed és true. Llença una
// excepció i s'esgota el temps d'espera.
```

```

public synchronized void waitForClose(final long
    timeoutMillis) throws Exception
{
    if(!closed) {
        wait(timeoutMillis);
        if(!closed) {
            throw new Exception("No es pot tancar el
MockResponseSender");
        }
    }
}

```

- Comentaris ambigus que no descriuen precisa o acuradament el que volen descriure.

El comentari de l'exemple anterior:

```
Mètode d'utilitat que retorna quan this.closed és true
```

És incorrecte, el mètode retorna **si** this.closed és true, no **quan** this.closed és true.

A.7. Classes

A.7.1. Notació

Els noms de les classes en Java segueixen les següents convencions:

- **Sempre** comencen en majúscula.
- Si el nom de la Classe és un nom compost, la primera lletra de cada paraula, incloent-hi la primera, s'escriu en majúscula.
- Les demés lletres s'escriuen totes en minúscula.
- No s'utilitza el guió baix per separar les diferents parts del nom d'una classe.

A.7.2. Noms de classes

Els noms de les classes i dels objectes han de ser noms comuns i amb significat específic, evitar els noms de significat dèbil i els verbs.

Noms incorretes.

```
Gestor, Controlador, Info, Data
```

Noms correctes.

```
PaginaWiki, Comprador, AddressParser
```

A.8. Excepcions

A.8.1. Excepcions genèriques

No capturar excepcions genèriques.

Incorrecte.

```
Widget readWidget(Reader in) throws Exception
```

Declarar o capturar les excepcions explícites que pot llançar el el bloc de codi o el mètode

Correcte.

```
Widget readWidget(Reader in) throws IOException,  
MalformedWidgetException
```

A.8.2. No amagar les excepcions

No amagar les excepcions simplement perquè el compilador no es queixi.

Incorrecte.

```
try {  
    double preu = in.readDouble();  
} catch (Exception e) {  
}
```

Correcte.

```
try {  
    double preu = in.readDouble();  
} catch (Exception e) {
```

```
    e.printStackTrace();
}
```

Apèndix B. Sistemes de representació de la informació numèrica

Sumari

B.1. Teorema fonamental dels nombres	925
B.2. Exemples de conversió a decimal	926
B.3. Conversió entre sistemes de numeració	926
B.4. Conversió d'una base b a base 10	926
B.5. Conversió de base decimal a base b	928
B.5.1. Per la part entera	928
B.5.2. Per la part decimal	928
B.5.3. Conversió de base b a base b'	929
B.6. Cas especial - Conversió de binari a Hexadecimal i viceversa	930
B.7. El sistema binari	931
B.8. Operacions bit a bit	933
B.9. Tipus d'operacions	933
B.10. Operacions bit a bit	934
B.10.1. NOT	934
B.10.2. AND	934
B.10.3. OR	935
B.10.4. XOR	935
B.10.5. Operacions de desplaçament	935

Un **sistema de numeració** permet representar la informació numèrica per mitjà de dades de significació numèrica i operar-hi, amb uns símbols i unes regles determinats.

Un **sistema de numeració** és el conjunt dels símbols i les normes que s'utilitzen per a la representació de la informació numèrica. En tot sistema de numeració hi ha una **base del sistema** que indica el nombre de símbols que podem utilitzar.



La notació és: **nombre_{base}**

Els **sistemes de numeració** són conjunts de dígits utilitzats per a representar quantitats. Així, hi ha els sistemes de numeració decimal, binari, octal, hexadecimal, romà, etc. Els quatre primers es caracteritzen per tenir una base (nombre de dígits diferents: 10, 2, 8 i 16, respectivament), mentre que el sistema romà no té base i la seva utilització resulta més complicada, tant amb nombres com en les operacions bàsiques (sumes, restes, multiplicacions i divisions).

Una **magnitud analògica**, com una tensió elèctrica o la velocitat pot tenir qualsevol valor dins d'un interval continu. Per exemple, la tensió a la sortida d'un micròfon podria ser dins de qualsevol valor entre 0 mV i 10 mV.

En la representació **digital**, les quantitats, representades per dígits, no poden tenir qualsevol valor, sinó sols valors discrets. Per exemple, un rellotge digital marca el temps del dia en forma de dígits decimals (hores, minuts i segons). Ara bé, el temps varia d'una manera contínua, però la posició de les busques del rellotge no varia d'una manera contínua, sinó a passos (segons). És a dir, el rellotge només pot donar valors discrets.

Un **sistema digital** és una combinació de dispositius (generalment electrònics) dissenyada per a manipular quantitats físiques o informació que estigui representada de manera digital (per exemple, els ordinadors, les calculadores digitals, els equips d'àudio i vídeo digital, els telèfons digitals, etc.).

La tecnologia digital utilitza molts sistemes de nombres. Els més comuns són:

Sistema decimal

El sistema decimal és el més conegut, ja que l'utilitzem contínuament. És format per deu símbols (0, 1, 2, 3, 4, 5, 6, 7, 8 i 9) amb els quals es pot

representar qualsevol quantitat numèrica. Aquest sistema també és conegut com de **base 10**. És un sistema de valor posicional.

Sistema binari

És el sistema de numeració que utilitzen internament els circuits digitals que configuren el maquinari dels ordinadors actuals. La **base és 2**, i això vol dir que es fan servir dos símbols per a representar la informació, que són **0 i 1**. Cadascun d'aquests símbols és conegut per la paraula **bit** (de l'anglès binary digit). Una combinació de vuit bits s'anomena **byte** (per exemple, 10011001).

Sistema octal

La **base és 8**. Per tant, els símbols que es poden aplicar són **0, 1, 2, 3, 4, 5, 6 i 7**.

Sistema hexadecimal

La **base és 16**. Per aquest motiu, es fan servir setze símbols, dels quals els deu primers són els nombres **0, 1, 2, 3, 4, 5, 6, 7, 8 i 9**, i els sis següents són les lletres **A, B, C, D, E, F** (els valors respectius són de 10 per a A, 11 per a B, 12 per a C, 13 per a D, 14 per a E i 15 per a F).

Per exemple, 19AC3 és un nombre en base hexadecimal.

B.1. Teorema fonamental dels nombres

Els sistemes de numeració són posicionals; és a dir, el valor relatiu de cada símbol és determinat pel seu valor absolut i per la seva posició relativa respecte de la coma decimal. Tot nombre es pot convertir al sistema decimal, utilitzant el **teorema fonamental dels nombres**.



El **teorema fonamental dels nombres** (TFN) diu que el valor **decimal** d'una quantitat expressada en altres sistemes de numeració s'expressa segons el polinomi següent: $\dots + x_4b^4 + x_3b^3 + x_2b^2 + x_1b^1 + x_0b^0 + x_{-1}b^{-1} \dots$.. en què el símbol b representa la base i x_n són els dígits del nombre.

B.2. Exemples de conversió a decimal

Exemple B.1. Conversió de binari a decimal

Donat el número 1011_2 , en calculem el valor decimal.

Apliquem el TFN:

$$1011_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 8 + 0 + 2 + 1 = 11_{10}$$



Recordeu que qualsevol nombre elevat a zero val 1.

Exemple B.2. Conversió de base 4 a decimal

Tenim la quantitat 3221_4 que s'expressa en base 4, aquesta base utilitza per a representar quantitats els dígits 0, 1, 2 i 3.

Quin serà el valor corresponent en el sistema decimal?

$$3 \cdot 4^3 + 2 \cdot 4^2 + 2 \cdot 4^1 + 1 \cdot 4^0 = 3 \cdot 64 + 2 \cdot 16 + 2 \cdot 4 + 1 = 233_{10}$$

B.3. Conversió entre sistemes de numeració

Moltes vegades, ens trobarem amb la necessitat de convertir quantitats d'un sistema de numeració a un altre. En aquests casos, cal tenir en compte tota una sèrie de regles.

B.4. Conversió d'una base b a base 10

Es pot fer mitjançant Teorema fonamental dels nombres. **Es fa de la mateixa manera tant si el nombre té decimals com si no en té.**

Exemple B.3. Conversió de base 5 a base 10

Representeu el 4123_5 en base 10 segons el teorema fonamental de la numeració.

$$4123_5 = 4 \cdot 5^3 + 1 \cdot 5^2 + 2 \cdot 5^1 + 3 \cdot 5^0 = 500 + 25 + 10 + 3 = 538_{10}$$

El mateix exercici anterior es pot fer amb una taula de la següent manera:

- Dibuixem una taula amb tantes columnes com díigits tingui el número que volem passar a base 10 i amb tres files:

- Posem el nombre a transformar a la fila de dalt i les potències de la base en la que està escrita el nombre a la fila de baix començant per la dreta:

4	1	2	3
125	25	5	1

- La última fila és el producte de les files anteriors:

4	1	2	3
125	25	5	1
500	25	10	3

- Finalment sumem els valors de l'última fila i això ens dona el resultat:

4	1	2	3	
125	25	5	1	
500	25	10	3	538₁₀

Exemple B.4. Conversió de base 2 a base 10 amb decimals

En el cas d'un **nombre amb decimals**, per exemple 10.11_2 , apliquem directament el teorema fonamental de la numeració.

$$10.11_2 = 1 \cdot 2^1 + 0 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} = 2 + 0 + 0.5 + 0.25 = 2.75_{10}$$



Recordeu que $a^{-b} = 1/a^b$

B.5. Conversió de base decimal a base b

Aquí hem de distingir entre la part entera i la part decimal del número.

B.5.1. Per la part entera

Hi ha diversos mètodes, però el més fàcil és el de **divisions successives per la base**. Es tracta d'anar dividint el nombre i els quocients successius entre la base fins que no podem més. La unió de tots els residus obtinguts escrits en ordre invers ens donarà el resultat buscat.

Exemple B.5. Conversió de base 10 a base 2

Passeu el nombre 75_{10} a base 2.

1. $75/2 = 37 + \text{residu } 1$
2. $37/2 = 18 + \text{residu } 1$
3. $18/2 = 9 + \text{residu } 0$
4. $9/2 = 4 + \text{residu } 1$
5. $4/2 = 2 + \text{residu } 0$
6. $2/2 = 1 + \text{residu } 0$
7. $1/2 = 0 + \text{residu } 1$

El resultat serà: 1001011_2

B.5.2. Per la part decimal

Per la part fraccionària hem de multiplicar successivament per la base b.

Com a l'altre factor de la primera multiplicació agafem la part fraccionària del nombre que volem representar en decimal.

Els factors dels altres productes seran les parts fraccionàries dels resultats de la multiplicació anterior.

El procés acaba quan la part fraccionària d'una multiplicació val 0. El resultat final s'obté juxtaposant la part entera dels resultats obtinguts.

Vegem-ho amb un exemple:

Exemple B.6. Conversió de base 10 a base 5

Expressa el nombre 0.44_{10} a base 5

$$1. \ 0.44 \cdot 5 = 2.2$$

$$2. \ 0.2 \cdot 5 = 1.0$$

El resultat serà: **21₅**

També és possible que aquest procés de multiplicacions no acabi mai, perquè es produeixi un cicle en els resultats. Quan ens n'adonem d'això, pararem el procés i expressarem el resultat com un nombre periòdic.

Exemple B.7. Conversió a un nombre periòdic

Expresseu el nombre $0'7_{10}$ en base 8.

$$1. \ 0.7 \cdot 8 = 5.6$$

$$2. \ 0.6 \cdot 8 = 4.8$$

$$3. \ 0.8 \cdot 8 = 6.4$$

$$4. \ 0.4 \cdot 8 = 3.2$$

$$5. \ 0.2 \cdot 8 = 1.6$$

$$6. \ 0.6 \cdot 8 = 4.8$$

I per tant $0.7_{10} = 0.5463146314631\dots_8$

Finalment, en el cas de tenir nombres amb part entera i part decimal, només hem d'operar les dues parts per separat i unir-les al final.

B.5.3. Conversió de base b a base b'

El mètode que s'ha de seguir pot ser el següent:

- Passar el nombre en base b a base 10.
- Passar el resultat anterior a la base b'.

Exemple B.8. Conversió de base 2 a base 3

Passeu el nombre 101_2 a base 3.

101_2 : és el 5 en base 10. Heu d'aplicar el TFN.

5_{10} : és el 12_3 . Dividint successivament per 3.

B.6. Cas especial - Conversió de binari a Hexadecimal i viceversa

Considerem l'equivalència entre els díigits hexadecimals i la seva representació binaria:

Hex	Bin
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Per passar d'hexadecimal a binari n'hi ha prou en substituir cada digit per la seva representació binaria.

Exemple B.9. Conversió d'hexadecimal a binari

$$\text{EF20A}_{16} = 1110\ 1111\ 0010\ 0000\ 1010_2$$

Per passar de binari a hexadecimal cal assegurar-se que tenim un nombre de díigits múltiple de 4 posant zeros a l'esquerra si cal i fer la conversió de cada grup de 4 díigits binaris a l'equivalent hexadecimal segons la taula anterior.

Exemple B.10. Conversió de binari a hexadecimal

$$1011010111_2 = 0010\ 1101\ 0111_2 = \text{2D7}_{16}$$

B.7. El sistema binari

El sistema binari o de base 2 és el sistema numèric posicional que utilitza dos símbols per representar els nombres. Els símbols són {0,1} i a cadascun d'ells se'ls anomena xifra binària o bit i es defineix com la unitat més petita de representació de la informació.

Les agrupacions de bits, dependent de la seva longitud, reben un nom:

4 bits	nibble
8 bits	byte
16 bits	mitja paraula
32 bits	paraula
64 bits	doble paraula

El bit i el byte són unitats de mesura de la quantitat d'informació. Per tant, s'utilitzen els mateixos prefixes que per qualsevol altre unitat per descriure els seus múltiples.

Fa alguns anys, aquesta unitat era suficient per mesurar la quantitat d'informació que hi havia en aquells moments, però avui resulta massa petita per als grans volums d'informació que es manipula i s'utilitzen prefixos per anomenar als múltiples del byte. S'utilitzen prefixos del SI o bé els prefixos binaris (IEC 60027-2).

En la pràctica popular, els prefixos binaris corresponen a nombres similars als factors indicats en el SI. Els primers són potències amb base 2, mentre que els prefixos del SI són potències amb base 10.

Aquesta diferència pot donar lloc a confusió a l'hora de mesurar quantitats de dades. Per tal d'evitar-ho, l'any 1998 la IEC va desenvolupar un estàndard on es varen definir unitats per a aquests prefixos binaris.

A continuació es mostren els dos sistemes de mesura:

Taula B.1. Taula Múltiples de bytes del SI i de la IEC

Prefix del SI (SI)		Prefix binari (IEC 60027-2)	
kilobyte kB	10^3 bytes	kibibyte KiB	2^{10} bytes
megabyte MB	10^6 bytes	mebibyte MiB	2^{20} bytes
gigabyte GB	10^9 bytes	gibibyte GiB	2^{30} bytes
terabyte TB	10^{12} bytes	tebibyte TiB	2^{40} bytes
petabyte PB	10^{15} bytes	pebibyte PiB	2^{50} bytes

En el món informàtic, el qual ja s'ha estès cap a la vida quotidiana, és molt habitual utilitzar els prefixos del SI quan realment haurien de fer servir els prefixos de la IEC. Per exemple, ens podem trobar especificacions tècniques que parlen de GB (gigabytes) quan realment haurien de dir GiB (gibibytes). Això passa perquè són prefixos de mesura molt similars. Fixeu-vos que 1 megabyte (1 MB) equival a 1.000.000 de bytes (106), i 1 mebibyte (1 MiB) equival a 1.048.576 bytes (220).

A mida que els prefixos augmenten (Gibi, Tebi,...), també s'incrementa la diferència entre tots dos sistemes. Així doncs cal parar atenció a la utilització correcta de les unitats.

La capacitat d'emmagatzematge és el camp d'aplicació habitual dels prefixos binaris i de les mesures informàtiques a partir del byte. En el camp de les mesures de les **velocitats de les comunicacions** és més comuna la utilització de prefixos del SI i d'unitats a partir del bit. Així doncs us podeu trobar amb la velocitat d'una xarxa indicada a 100 megabits per segon (100 Mbps).



En general quan es treballa en Bytes les equivalencies entre els prefixes són en potències de 2.

Quan es treballa en bits s'acosuma a treballar en potències de 10

Taula B.2. Treballant en Bytes

prefix	valor
1 quilobyte (KB)	1024 bytes
1 megabyte (MB)	1024 KB
1 gigabyte (GB)	1024 MB
1 terabyte (TB)	1024 GB
1 petabyte (PB)	1024 TB

Taula B.3. Treballant en bits

1 quilobit (Kbit)	1000 bits
1 megabit (Mbit)	1000 Kbit
1 gigabit (Gbit)	1000 Mbit
1 terabit (Tbit)	1000 Tbit
1 petabit (PB)	1000 Tbit

B.8. Operacions bit a bit

Una operació bit a bit (en anglès **bitwise operation**) opera sobre nombres binaris a nivell dels seus bits individuals.

És una acció primitiva ràpida i és suportada directament pels processadors.

En processadors simples de baix cost, les operacions de bit a bit, juntament amb els d'addició i substracció, són substancialment més ràpides que la multiplicació i la divisió, mentre que en els moderns processadors d'alt rendiment usualment les operacions es realitzen a la mateixa velocitat.

B.9. Tipus d'operacions

Es poden distingir fins a tres tipus diferents d'operacions a nivell de bit:

Operacions bit a bit

Executen les operacions lògiques AND, OR, XOR, NOT, etc., sobre els bits individuals dels operands.

Operacions de desplaçament

Desplacen els bits dels operands cap a la dreta o cap a l'esquerra una o més posicions.

Operacions de rotació

Roten els bits de l'operand cap a la dreta o cap a l'esquerra una o més posicions. Es pot, o no, usar el flag de ròssec com un bit addicional en la rotació.

B.10. Operacions bit a bit

B.10.1. NOT

El NOT bit a bit, o bitwise, o complement, és una operació unaria que realitza la negació lògica a cada bit, invertint els bits del nombre, de tal manera que els zeros es converteixen en 1 i viceversa. Per exemple:

```
NOT 0111 (decimal 7)
= 1000 (decimal 8)
```

B.10.2. AND

El AND bit a bit, o bitwise, presa dos nombres enters i realitza l'operació AND lògica a cada parell corresponent de bits. El resultat en cada posició és 1 si el bit corresponent dels dos operands és 1, i 0 en cas contrari, per exemple:

```
0101 (decimal 5)
AND 0011 (decimal 3)
= 0001 (decimal 1)
```

El AND pot ser usat per filtrar determinats bits, permetent que uns bits passin i els altres no.

B.10.3. OR

Una operació OR de bit a bit, o bitwise, presa dos nombres enters i realitza l'operació OR inclusiu a cada parell corresponent de bits. El resultat en cada posició és 1 si el bit corresponent d'una dels dos operands és 1, i 0 si tots dos bits són 0, per exemple:

```
0101  
OR 0011  
= 0111
```

B.10.4. XOR

El XOR bit a bit, o bitwise, presa dos nombres enters i realitza l'operació OR exclusiu a cada parell corresponent de bits. El resultat en cada posició és 1 si el parell de bits són diferents i zero si el parell de bits són iguals.

```
0101  
XOR 0011  
= 0110
```

B.10.5. Operacions de desplaçament

En aquestes operacions els díigits (bits) són moguts, o desplaçats, cap a l'esquerra o cap a la dreta.

Els registres en un processador d'ordinador tenen una amplada fixa, així que alguns bits "seran desplaçats cap a fora" ("shifted out"), és a dir, "surten" del registre per un extrem, mentre que el mateix nombre de bits són "desplaçats cap a dintre" ("shifted in"), és a dir, "entren" per l'altre extrem, les diferències entre els operadors de desplaçament de bits estan en com aquests determinen els valors dels bits que entren al registre (desplaçament cap a dins) (shifted-in).

Per exemple. Si es té en un registre de 8 bits el valor 10110011, i es fa un desplaçament cap a l'esquerra d'un bit, tots els bits es mouen una posició cap a l'esquerra, el bit de l'esquerra es perd i entra un bit zero de farciment pel costat dret. En un desplaçament d'un bit cap a la dreta passa alguna cosa semblant, el bit de la dreta es perd i el de l'esquerra s'omple amb un zero:

Operacions de desplaçament

10110011 desplaçament 1 <-> 0110011 <-> 0 01100110 desplaçament cap a l'esquerra	10110011 0 --> 1011001 --> 1 01011001 desplaçament cap a la dreta	<-> Bits abans del Desplaçament <-> Bits després del
---	---	--

Apèndix C. Representació de dades a la memòria de l'ordinador

Sumari

C.1. Representació d'enters	938
C.2. Enters sense signe	938
C.3. Enters amb signe	939
C.3.1. Representació en signe-valor absolut	939
C.3.2. Representació en complement a 1	941
C.3.3. Representació en complement a 2	943
C.3.4. Descodificar nombres representats en complement a 2	947
C.4. Big Endian vs. Little Endian	948
C.5. Representació de nombres en coma flotant	949
C.5.1. L'estàndard normalitzat IEEE 754	950
C.5.2. Forma de-normalitzada	953
C.6. Codificació de caràcters	954
C.6.1. ASCII	954
C.6.2. ISO 8859	956
C.6.3. UNICODE	956
C.6.4. UTF-8	957
C.6.5. Codificació del caràcter de final de línia	958

Els ordinadors utilitzen una quantitat fixa de bits per representar un peça d'informació. Aquesta pot ser un **número**, un **caràcter** o d'altres.

Una localització de memòria de n bits pot representar fins a 2^n valors diferents. Per exemple, si prenem una localització de memòria de 3 bits es poden representar 8 entitats diferents, per tant es podria representar els nombres del 0 al 7, els nombres del 8881 al 8888, els caràcters de la A a la H o 8 noms propis de persona diferents.

Els enters, per exemple, es poden representar amb 8, 16, 32 o 64 bits, el programador es qui tria la longitud apropiada en cada cas cosa que imposa restriccions en el rang de valors que poden ser representats.

A part de la longitud, un enter es pot representar en diferents esquemes de representació, per exemple, un enter de 8 bits amb signe té un rang entre 0 i 255 i un enter sense signe té un rang entre -128 i 127

És important notar que **una localització de memòria només emmagatzema un patró binari**, la interpretació del significat d'aquest patró depèn únicament del programador. Quan fem referència a la interpretació d'un patró binari diem que fem referència a una codificació de dades.

C.1. Representació d'enters

Els ordinadors utilitzen un nombre fix de bits per representar un enter. Les longituds més habituals són 8, 16, 32 i 64 bits.

A part de la longitud, existeixen diferents esquemes de representació:

- Enters sense signe
- Enters amb signe
 - Representació en signe-valor absolut
 - Representació en complement a 1
 - Representació en complement a 2

C.2. Enters sense signe

Els enters sense signe poden representar el zero i enters positius.

El valor d'un enter sense signe s'interpreta directament com la **representació en base 2** de l'enter, afegint tants zeros a l'esquerra com faci falta.

Per exemple:

Exemple C.1. Representació d'un número en 32 bits

La representació del número 9 en 32 bits serà:

$$9_{10} = 1 * 2^3 + 1 * 2^0 = 1001_2$$

I en 32 bits: **00000000 00000000 00000000 00001001**

C.3. Enters amb signe

Els enters amb signe poden representar el zero i tots els enters tant positius com negatius.

Existeixen tres esquemes de representació per representar els enters amb signe:

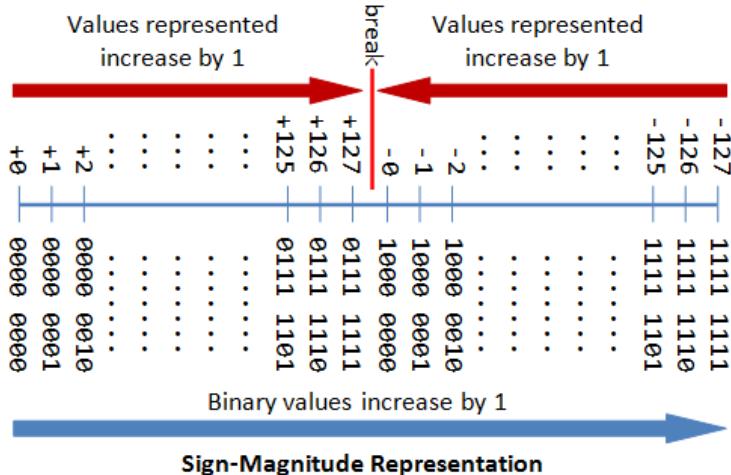
- Representació en signe-valor absolut
- Representació en complement a 1
- Representació en complement a 2

Els tres esquemes anteriors **reserven el bit més significatiu ¹ per representar el signe**, utilitzen el 0 per representar els nombres positius i l'1 per representar els nombres negatius. No obstant, **la representació del valor absolut del nombre es codifica de forma diferent en cadascun d'ells**.

C.3.1. Representació en signe-valor absolut

Aquesta codificació utilitza el bit més significatiu per representar el signe i els bits restants representen el valor absolut de l'enter com si es tractés d'un enter sense signe.

¹ El bit situat més a l'esquerra



Per exemple:

Exemple C.2. Representació en signe-magnitud

Considerem les següents representacions d'enters de 8 bits:

$$01000001_2 = 65_{10}$$

$$10000001_2 = -1_{10}$$

$$00000000_2 = 0_{10}$$

$$10000000_2 = -0_{10}$$

Els inconvenients de treballa amb aquest esquema són:

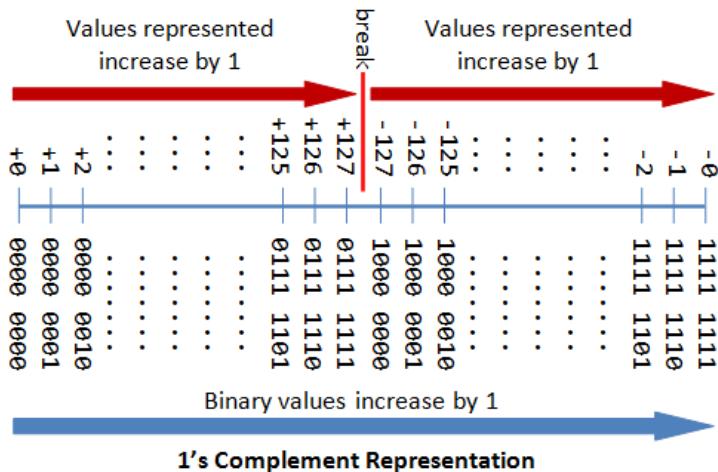
- Existeixen dues representacions del número 0 (00000000_2 i 10000000_2) cosa que fa que alguns càlculs siguin ineficients.
- El tractament dels nombres positius i negatius s'ha de realitzar per separat, per exemple l'algorisme utilitzat per sumar dos nombres positius és diferent que l'utilitzat per sumar un nombre negatiu i un nombre positiu.

C.3.2. Representació en complement a 1

De la mateixa manera que en el cas anterior, aquesta representació pren el bit més significatiu com a bit de signe, un valor de 0 representa un nombre positiu enter i un valor de 1 representa un nombre negatiu enter.

Els bits restants representen la magnitud de l'enter de la següent manera:

- Si l'enter és positiu, el valor absolut de l'enter és igual a la magnitud del patró de bits del nombre exceptuant-ne el primer, que és el signe.
- Si l'enter és negatiu, el valor absolut de l'enter és igual a la magnitud del patró de bits del nombre, exceptuant-ne el primer, fent una operació **NOT**, és a dir invertint cadascun dels bits.



Per exemple:

Exemple C.3. Representació en complement a 1

Considerem la següent representació d'un enter de 8 bits $0\ 100\ 0001_2$

- Bit de signe: 0 # positiu
- Valor absolut: $100\ 0001_2 = 65_{10}$
- Per tant l'enter és $+65_{10}$

Exemple C.4. Representació en complement a 1

Considerem la següent representació d'un enter de 8 bits **1 000 0001₂**

- Bit de signe: 0 # negatiu
- Valor absolut: Ca1(000 0001) = 111 1110₂ = 126₁₀
- Per tant l'enter és **-126₁₀**

Exemple C.5. Representació en complement a 1

Considerem la següent representació d'un enter de 8 bits **0 000 0000₂**

- Bit de signe: 0 # positiu
- Valor absolut: 000 0000₂ = 0₁₀
- Per tant l'enter és **+0₁₀**

Exemple C.6. Representació en complement a 1

Considerem la següent representació d'un enter de 8 bits **1 111 1111₂**

- Bit de signe: 0 # negatiu
- Valor absolut: Ca1(111 1111) = 000 0000₂ = 0₁₀
- Per tant l'enter és **-0₁₀**

Els inconvenients de treballa amb aquest esquema són:

- Existeixen dues representacions del número 0 (00000000₂ i 11111111₂) cosa que fa que alguns càlculs siguin ineficients.
- Un carreteig al bit més significatiu fa que el resultat sigui incorrecte i que s'hagi de tenir en compte. Per tant els càlculs es ralentitzen.

Per exemple:

La següents sumes es realitzen automàticament:

Exemple C.7. Suma en complement a 1

$$\begin{array}{r}
 0001\ 0110 \quad 22 \\
 +\ 0000\ 0011 \quad 3 \\
 \hline
 0001\ 1001 \quad 25
 \end{array}$$

Exemple C.8. Resta en complement a 1

$$\begin{array}{r}
 0000\ 0001 \quad 1 \\
 +\ 1111\ 1101 \quad - 2 \\
 \hline
 1111\ 1110 \quad - 1
 \end{array}$$

La següents sumes es realitzen amb errors, caldrà sumar o restar el bit de carreteig segons el cas:

Exemple C.9. Resta en complement a 1 amb errors

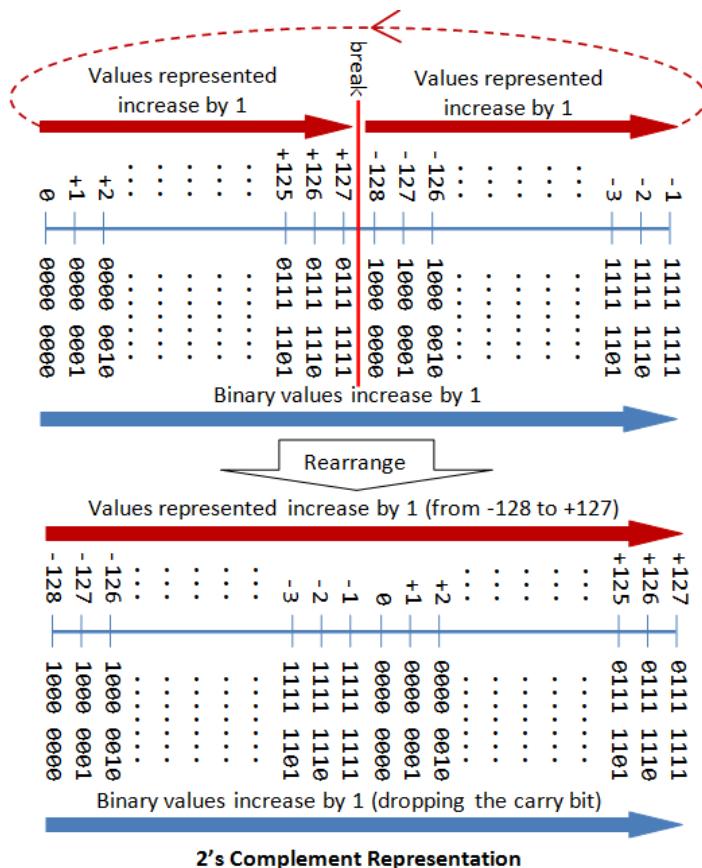
$$\begin{array}{r}
 0001\ 0110 \quad 22 \\
 +\ 1111\ 0001 \quad - 14 \\
 \hline
 (1)\ 0000\ 0101 \quad 09 \text{ (Error, caldrà restar el bit de carreteig)}
 \end{array}$$

C.3.3. Representació en complement a 2

Altre cop, aquest esquema utilitza el bit més significatiu per representar el signe, 0 positiu, 1 negatiu. Els bits restants representen la magnitud de l'enter de la següent manera:

- Si l'enter és positiu, el valor absolut de l'enter és igual a la magnitud del patró de bits del nombre exceptuant-ne el primer, que és el signe.
- Si l'enter és negatiu, el valor absolut de l'enter és igual a la magnitud del patró de bits del nombre, exceptuant-ne el primer, fent una operació NOT i sumant 1.

Aquest esquema respon a la idea de que el nombre invers de N és aquell que sumat a N dona 0. Tenint en compte que la quantitat de bits per representar un enter és fixa.



Exemple C.10. Representació en complement a 2

Considerem la següent representació d'un enter de 8 bits $0\ 100\ 0001_2$

- Bit de signe: 0 # positiu
- Valor absolut: $100\ 0001_2 = 65_{10}$
- Per tant l'enter és $+65_{10}$

Exemple C.11. Representació en complement a 2

Considerem la següent representació d'un enter de 8 bits $1\ 000\ 0001_2$

- Bit de signe: 1 # negatiu
- Valor absolut: $Ca2(000\ 0001) = 111\ 1111_2 = 127_{10}$
- Per tant l'enter és -127_{10}

Exemple C.12. Representació en complement a 2

Considerem la següent representació d'un enter de 8 bits $0\ 000\ 0000_2$

- Bit de signe: 0 # positiu
- Valor absolut: $000\ 0000_2 = 0_{10}$
- Per tant l'enter és $+0_{10}$

Exemple C.13. Representació en complement a 2

Considerem la següent representació d'un enter de 8 bits $1\ 111\ 1111_2$

- Bit de signe: 1 # negatiu
- Valor absolut: $Ca2(111\ 1111) = 000\ 0001_2 = 1_{10}$

Els esquemes de numeració actuals utilitzen la representació complement a 2 pels enters amb signe

La representació de complement a 2 té una sola representació del zero i les operacions de suma i resta es poden tractar de la mateixa manera.

Exemple C.14. Suma en complement a 2

Considerem que treballem amb aritmètica de 8 bits.

$$\begin{array}{rcl}
 65_{10} & -> & 0100\ 0001_2 \\
 + 5_{10} & -> & + 0000\ 0101_2 \\
 \hline
 70_{10} & -> & 0100\ 0110_2
 \end{array}$$

Exemple C.15. Suma en complement a 2

Considerem que treballem amb aritmètica de 8 bits.

$$\begin{array}{rcl}
 65_{10} & -> & 0100\ 0001_2 \\
 - 5_{10} & -> & + 1111\ 1011_2 \\
 \hline
 60_{10} & -> & (1)0011\ 1100_2
 \end{array}$$

Exemple C.16. Suma en complement a 2

Considerem que treballem amb aritmètica de 8 bits.

$$\begin{array}{rcl}
 -65_{10} & -> & 1011\ 1111_2 \\
 + -5_{10} & -> & + 1111\ 1011_2 \\
 \hline
 -70_{10} & -> & (1)1011\ 1010_2
 \end{array}$$

El fet de tenir un nombre fixat de bits implica que el complement a 2 d'un enter amb signe té un determinat rang.

Per exemple, amb 8 bits, el rang dels nombres enters en complement a 2 és de -128 a 127. Per tant, durant la suma és important **comprovar si el resultat excedeix el rang permès o no**, en d'altres paraules, comprovar si s'ha produït una situació de **overflow** o de **underflow**.

Exemple C.17. Suma en complement a 2

Considerem que treballem amb aritmètica de 8 bits.

$$\begin{array}{rcl} 127_{10} & -> & 0111\ 1111_2 \\ + \quad 2_{10} & -> & +\ 0000\ 0010_2 \\ \hline 129_{10} & -> & 1000\ 0001_2 = -127_{10} \text{ (error overflow)} \end{array}$$

Exemple C.18. Suma en complement a 2

Considerem que treballem amb aritmètica de 8 bits.

$$\begin{array}{rcl} -125_{10} & -> & 1000\ 0011_2 \\ + \quad -5_{10} & -> & +\ 1111\ 1011_2 \\ \hline -130_{10} & -> & 0111\ 1110_2 \sim +126_{10} \text{ (error underflow)} \end{array}$$

C.3.4. Descodificar nombres representats en complement a 2

Una manera ràpida de descodificar nombres expressats en complement a 2 és la següent:

- Si el nombre és positiu, el valor absolut del nombre és la seva representació binaria (traient el bit de signe)
- Si el nombre és negatiu, S'extreu el bit de signe, es busca des de la dreta el primer 1 i s'intercanvien tots els bits a l'esquerra d'aquest primer 1, per exemple:

Exemple C.19. Passar un nombre de complement a 2 a decimal

Considerem que treballem amb aritmètica de 8 bits.

Prenem el nombre $1100\ 0100_2$

- Mirem signe: $1100\ 0100_2 \rightarrow$ negatiu
- Busquem primer 1 des de la dreta $100\ 0100_2$
- Intercanviem tots els bits a l'esquerra de l'anterior: $011\ 1100_2 = 60_{10}$
- Per tant el nombre és -60_{10}

C.4. BigEndian vs. LittleEndian

Els ordinadors actuals emmagatzemen **un byte** de dades a cada localització de memòria. Per tant, un enter de 32 bits ocupa **4 adreces de memòria**.

El terme **endian** fa referència a l'ordre d'emmagatzematge dels bytes a la memòria de l'ordinador.

bigEndian

Els bytes **més** significatius s'emmagatzemen en primer lloc, a les adreces de memòria més baixes.

littleEndian

Els bytes **menys** significatius s'emmagatzemen en primer lloc, a les adreces de memòria més baixes.

Exemple C.20. BigEndian vs LittleEndian

Considerem l'enter de 32 bits 12345678_{16}

- En **bigEndian** s'emmagatzemaria: 00010010 00110100 01010110 01111000
- En **littleEndian** s'emmagatzemaria: 01111000 01010110 00110100 00010010

C.5. Representació de nombres en coma flotant

Per representar els nombres no enters s'utilitzen els sistemes anomenats de **coma flotant**.

Un nombre en coma flotant pot representar nombres molt grans ($1.45 \cdot 10^{96}$) o nombres molt petits ($1.45 \cdot 10^{-96}$), també pot representar nombres negatius, grans ($-1.45 \cdot 10^{96}$) i petits ($-1.45 \cdot 10^{-96}$).

Un nombre en coma flotant se sol representar en notació científica, amb una **base** (B), una **mantissa** (M), i un **exponent** (E).

Els nombres decimals utilitzen la base 10 i es representen com $\pm M \cdot 10^{\pm E}$. Els nombres binaris es representarien com $\pm M \cdot 2^{\pm E}$

Cal tenir en compte que la representació d'un nombre en coma flotant **no és única**. Per exemple, el nombre **12.34** es pot representar com **$1.234 \cdot 10^1$** , **$0.1234 \cdot 10^2$** , **$0.01234 \cdot 10^3$** , etc...

Per què tot funcioni, caldrà triar una de les representacions possibles com a representació canònica. Definirem la **forma normalitzada d'un nombre en coma flotant** com aquella representació del nombre que té **un únic index abans de la coma decimal**.

Per exemple, la forma formalitzada del nombre **12.34** serà **$1.234 \cdot 10^1$** .

Si ho pensem en binari, la forma normalitzada del nombre **1010.1011_2** serà **$1.0101011 \cdot 2^3$** .



És important notar que els nombres en coma flotant tenen errors de precisió quan es representen amb un nombre fix de bits. La raó és que hi ha infinitis valors reals, fins i tot en un rang petit com 0.1 a 0.1, i un patró binari de per exemple 32 bits permet representar només 2^{32} valors.

En general, en cas que un nombre no es pugui representar correctament, es prendrà la millor aproximació, resultant en una pèrdua de precisió.



També és important notar que l'aritmètica en coma flotant és molt **menys** eficient que l'aritmètica entera.

Exemple C.21. Errors de presisió en aritmètica flotant

Suposem que treballem amb decimal i tenim **tres xifres** per indicar el valor del nombre i una per l'exponent. Situem la coma al darrera de l'última xifra de la mantissa.

El nombre més gran que podem expressar és el **$999 \cdot 10^9$** i el més petit **$-999 \cdot 10^9$** (estem suposant que el signe no ens ocupa una xifra).

Fixeu-vos però que no és possible representar un nombre tant gran si necessitem que tingui xifres decimals. Per representar nombres amb decimals haurem d'utilitzar un exponent negatiu.

El nombre **99,9** es representa **$999 \cdot 10^{-1}$** i és el nombre més gran que podem representar amb una xifra decimal.

El nombre positiu més proper a 0 que podem representar és **$001 \cdot 10^9$** el que correspon al **0,000000001**.

Fixeu-vos també que tenim el mateix problema pels nombres propers a 0 que pels nombres més grans:

- només podem representar un nombre tan petit com aquest perquè totes les seves primeres xifres són 0.
- No podem, per exemple, representar el nombre 0,000001001 amb aquest sistema.

C.5.1. L'estàndard normalitzat IEEE 754

De totes les variants possibles de mètodes de coma flotant, l'estàndard IEEE 754 és el que s'utilitza per la representació dels nombres reals en un ordinador.

Aquest estàndard té dues variants, la de precisió simple que utilitza 32 bits per a la representació i la de precisió doble que n'utilitza 64. Nosaltres veurem només el de precisió simple.

L'estructura del nombre és la següent:

- 1 bit pel signe.
- 8 bits per l'exponent.
- 23 bits per la mantissa.

Tenint en compte que:

- El valor del nombre s'expressa en signe-magnitud, però entremig del signe i de la mantissa s'hi col·loca l'exponent. Aquest ordre s'utilitza per fer més ràpida la comparació entre nombres.
- Per calcular l'exponent es pren la versió **normalitzada** del nombre. És a dir, l'exponent s'ajusta de manera que la coma quedi al darrera del primer 1 de la mantissa. **Aquest bit a 1 no s'escriu (és per tant implícit)**.
- L'exponent s'expressa en excés a 127. Aquesta representació té avantatges similars a la de complement a 2, però facilita la comparació.

Excés a 127

La representació en excés a M consisteix a escriure els nombres en la seva representació binària sense signe, però sumant M al nombre abans.

Exemple C.22. Representació en excés 127

Representa els nombres 67 i -125 en excés a 127 i utilitzant 8 bits.

- $67 + 127 = 194 = 11000010_2$
- $-125 + 127 = 2 = 00000010_2$

Per passar de representació en excés a M a decimal només hem de passar el nombre a decimal i sumar-li M.

Exemple C.23. Conversió d'excés 127 a decimal

El nombre 11011010 està representat en excés a 127. De quin nombre es tracta?

$$11011010_2 = 128 + 64 + 16 + 8 + 2 = 218 \rightarrow 218 - 127 = 91_{10}$$

El nombre 11011010 és el **91₁₀**.

Exemple C.24. Estàndard normalitzat IEEE 754 (32 bits)

Prenem el nombre **1 1000 0001 011 0000 0000 0000 0000 0000**

- Signe: 1
- Exponent: 1000 0001
- Mantissa: 011 0000 0000 0000 0000 0000

En la forma normalitzada hem d'entendre que la mantissa té un 1. implícit i per tant la mantissa real de l'exemple seria:

- Mantissa: **1.011 0000 0000 0000 0000 0000** = $1 + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} = 1.375_{10}$
- Signe: 1 implica negatiu
- Exponent: En excés 127 per tant $1000\ 0001_2 - 127_{10} = 2_{10}$

Per tant, el nombre representat és:

$$-1.375 \cdot 2^2 = -5.5_{10}$$

C.5.2. Forma de-normalitzada

La forma normal d'un nombre en coma flotant té un seriós problema, el fet de tenir sempre un 1 implícit a la mantissa **fa impossible poder representar el número zero**.

Per tractar aquest cas els nombres **amb un exponent igual a 0** no es normalitzen i per tant es considera un 0. implícit a la mantissa i un **exponent de -126**.

En la forma de-normalitzada es poden representar nombres molt petits tant negatius com positius.

Exemple C.25. Representació IEEE-754 de un nombre de 32 bits

Suposem que la representació IEEE-754 de un nombre de 32 bits és **0 0000000 011 0000 0000 0000 0000 0000**

$$0.011 = 1 \cdot 2^{-2} + 1 \cdot 2^{-3} = 0.375_{10}$$

$$\text{El nombre és: } 0.375 \cdot 2^{-126} = -4.4 \cdot 10^{-39}$$

Exemple C.26. Representació IEEE-754 de un nombre de 32 bits

Suposem que la representació IEEE-754 de un nombre de 32 bits és **0 10000000 110 0000 0000 0000 0000 0000**

- Signe: positiu
- Exponent: $10000000_2 = 128_{10}$ (forma normalitzada)
- Mantissa: $1.11_2 = 1 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} = 1.75_{10}$

El nombre és: $+1.75 \cdot 2^1 = +3.5_{10}$

Exemple C.27. Representació IEEE-754 de un nombre de 32 bits

Suposem que la representació IEEE-754 de un nombre de 32 bits és **1 0111110 100 0000 0000 0000 0000 0000**

- Signe: negatiu
- Exponent: $0111110_2 = 126_{10}$ (forma normalitzada)
- Mantissa: $1.1_2 = 1 + 1 \cdot 2^{-1} = 1.5_{10}$

El nombre és: $+1.5 \cdot 2^{-1} = +0.75_{10}$

C.6. Codificació de caràcters

A la memòria de l'ordinador, els caràcters es codifiquen o es representen utilitzant un **esquema de codificació de caràcters** també anomenat **charset** o **code page**.



És important notar que l'esquema de representació ha de ser conegit previament abans de poder interpretar un patró binari que representa un caràcter.

C.6.1. ASCII

L'ASCII (American Standard Code for Information Interchange, Codi Estàndard Americà per a l'Intercanvi d'Informació) és un joc de caràcters que assigna valors numèrics de 7 bits a les lletres, xifres i signes de puntuació.

Inicialment emprava un bit addicional (bit de paritat) que s'usava per detectar errors en la transmissió. En l'actualitat defineix codis per 33 caràcters no imprimibles, dels quals la majoria són caràcters de control obsolets que tenen efecte sobre com es processa el text, i 95 caràcters imprimibles.

Gairebé tots els sistemes informàtics actuals utilitzen el codi ASCII o una extensió compatible per a representar textos.

Taula ASCII (7 bits).

Dec	Char	Dec	Char	Dec	Char	Dec
Char		Char		Char		Char
0	NUL (null)	32	SPACE	64	@	96 `
1	SOH (start of heading)	33	!	65	A	97 a
2	STX (start of text)	34	"	66	B	98 b
3	ETX (end of text)	35	#	67	C	99 c
4	EOT (end of transmission)	36	\$	68	D	100 d
5	ENQ (enquiry)	37	%	69	E	101 e
6	ACK (acknowledge)	38	&	70	F	102 f
7	BEL (bell)	39	'	71	G	103 g
8	BS (backspace)	40	(72	H	104 h
9	TAB (horizontal tab)	41)	73	I	105 i
10	LF (NL line feed, new line)	42	*	74	J	106 j
11	VT (vertical tab)	43	+	75	K	107 k
12	FF (NP form feed, new page)	44	,	76	L	108 l
13	CR (carriage return)	45	-	77	M	109 m
14	SO (shift out)	46	.	78	N	110 n
15	SI (shift in)	47	/	79	O	111 o
16	DLE (data link escape)	48	0	80	P	112 p
17	DC1 (device control 1)	49	1	81	Q	113 q
18	DC2 (device control 2)	50	2	82	R	114 r
19	DC3 (device control 3)	51	3	83	S	115 s
20	DC4 (device control 4)	52	4	84	T	116 t
21	NAK (negative acknowledge)	53	5	85	U	117 u
22	SYN (synchronous idle)	54	6	86	V	118 v
23	ETB (end of trans. block)	55	7	87	W	119 w
24	CAN (cancel)	56	8	88	X	120 x
25	EM (end of medium)	57	9	89	Y	121 y
26	SUB (substitute)	58	:	90	Z	122 z
27	ESC (escape)	59	;	91	[123 {
28	FS (file separator)	60	<	92	\	124
29	GS (group separator)	61	=	93]	125 }
30	RS (record separator)	62	>	94	^	126 ~

31 US (unit separator)

63 ?

95 _

127 DEL

C.6.2. ISO 8859

L'ISO 8859 és una sèrie de fulls de codis estàndards numerats (ISO 8859-1, ISO 8859-2, etc.) de 8 bits. Cadascun del fulls engloba un conjunt d'idiomes propers i tots ells coincideixen amb l'ASCII en els primers símbols.

Per exemple, l'ISO-8859-1 (latin-1 western european), probablement el més estès, engloba els caràcters de la major part d'idiomes de l'Europa occidental: anglès, francès (â, ê...), alemany (ß), portuguès (ç), español (ñ), etc. L'ISO-8859-15 incorpora el símbol de l'euro (€) a aquest conjunt.

Tot i que encara s'utilitza molt en les pàgines web i algunes distribucions de Linux, actualment està prenent més força la codificació UTF-8.

C.6.3. UNICODE

L'Unicode és un estàndard internacional de codificació de caràcters en suports informàtics. El seu objectiu és proporcionar el mitjà per a permetre emmagatzemar qualsevol text que es desitgi. Això inclou qualsevol mena de forma d'escriptura que es faci servir actualment, moltes formes d'escriptura conegeudes només pels estudiosos i altra mena de símbols com ara els símbols matemàtics o lingüístics. La versió 5.1 conté 100.713 caràcters d'alfabets, sistemes ideogràfics i col·leccions de símbols (matemàtics, tècnics, musicals, icones...).

L'Unicode és un projecte que pretén reemplaçar tota mena de conjunt de caràcters existent. Avui en dia, l'Unicode es considera el conjunt de caràcters més complet i ha esdevingut l'opció preferida en la internacionalització de programari en entorns multilingües. Molts estàndards recents i programari bàsic han adoptat Unicode per a representar text.

Els caràcters s'identifiquen mitjançant un nombre o punt de codi i el seu nom o descripció. L'espai de codis té 1.114.112 posicions possibles (0x10FFFF). Els punts de codi es representen en notació hexadecimal, afegint el prefix U+. El valor hexadecimal es completa amb zeros fins a 4 díigits hexadecimals si cal; si és de longitud major que 4 díigits, no s'afegeixen zeros.

C.6.4. UTF-8

L'UTF-8 és un format de codificació de caràcters de longitud variable que permet representar qualsevol caràcter Unicode i que coincideix amb l'ASCII en la codificació dels símbols comuns.

Rang de codi Unicode	UTF-8	Notes
000000 - 00007F	0xxxxxx	Rang equivalent a l'ASCII. Símbols d'un únic byte on el bit més significatiu és 0
000080 - 0007FF	110xxxxx 10xxxxxx	Símbols de dos bytes. El primer byte comença amb 110, el segon byte comença amb 10
000800 - 00FFFF	1110xxxx 10xxxxxx 10xxxxxx	Símbols de tres bytes. El primer byte comença amb 1110, els bytes següents comencen amb 10
010000 - 10FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	Símbols de quatre bytes. El primer byte comença amb 11110, els bytes següents comencen amb 10

L'UTF-8 és de les codificacions més utilitzades actualment en web i correu electrònic, perquè permet codificar tots els idiomes amb la mateixa codificació sense un gran augment en la longitud del missatge.

Encara hi ha algun component que pot treballar amb la seves codificacions anteriors, com la CP-1252 (utilitzada per les versions amb idiomes de l'Europa occidental i no acceptada com estàndard), cosa que causa diversos problemes en l'intercanvi d'informació des d'aquest o cap aquest sistema.

C.6.5. Codificació del caràcter de final de línia

Les aplicacions i els sistemes operatius normalment representen en caràcter de nova línia amb un o dos codis de control:

Els sistemes basats en ASCII o compatibles utilitzen LF (salt de línia, \n), CR (retorno de carro, \r), o CRLF (CR seguit de LF):

LF

Unix i sistemes tipus Unix, Linux, Mac OS X i altres.

CR+LF

DOS, Microsoft Windows i altres.

CR

Mac OS fins a la versió 9.

Els valors numèrics utilitzats normalment són:

CR

decimal 13, hexadecimal 0D

LF

decimal 10, hexadecimal 0A

El CRLF no és més que un rere l'altre: 0D 0A en hexadecimal.

Apèndix D. Generació de nombres aleatoris en Java

Sumari

D.1. Nombres aleatoris	960
D.2. Classe Random	960

D.1. Nombres aleatoris

Tot i que usualment utilitzem el terme nombres aleatoris, Java realment genera nombres pseudoaleatoris, això vol dir que Java pot generar una seqüència de nombres aparentment aleatoris però aquesta seqüència està inicialitzada per un valor numèric anomenat **llavor**.

Si s'utilitza la mateixa llavor per generar una seqüència de nombres aleatoris s'obtindrà la mateixa seqüència

Java inclou un objecte anomenat **Random** que permet generar seqüències de diferents tipus de nombres aleatoris.

Veurem com crear nombres aleatoris de tipus int i double amb una distribució uniforme, és a dir, amb la mateixa probabilitat d'aparició.

D.2. Classe Random

1. Per utilitzar la classe Random primer caldrà importar el paquet.

```
import java.util.Random;
```

2. Caldrà crear un objecte de tipus Random que pugui generar els nombres aleatoris que necessitem.

```
Random generadorAleatori = new Random();
```

Ens pot interessar especificar una **llavor** per obligar a que les seqüències de nombres aleatòries siguin les mateixes per a cada execució, en aquest faríem:

```
int llavor = 8; ①  
Random generadorAleatori = new Random(llavor);
```

- ① En funció del número de la llavor tindrem una seqüència de nombres aleatoris o una altra.
3. Podem obtenir nombres aleatoris a partir de la variable **generadorAleatori** creada anteriorment de la següent manera:

- Si es vol un nombre aleatori entre tot el rang de ints.

```
int nombreAleatori = generadorAleatori.nextInt();
```

- Si es vol un nombre aleatori entre 0.0 i 1.0

```
double nombreAleatori = generadorAleatori.nextDouble();
```

- Si es vol un nombre aleatori de 0 a n-1.

- En el cas de **nombres enters**, el mètode **nextInt** permet indicar el número més gran de l'interval.

```
int nombreAleatori = generadorAleatori.nextInt(9); // de 0 - 8
```

- En el cas de `nextDouble` s'ha de fer multiplicant el resultat de `nextDouble` per una potència de 10.

```
double nombreAleatori = generadorAleatori.nextDouble(); //  
Torna un valor entre 0 i 1  
// nombreAleatori * 10; // Torna un valor entre 0 i  
9.9999...999  
// nombreAleatori * 100; // Torna un valor entre 0 i  
99.999...999
```

- Si es volen diferents rangs es pot reescalar el resultat.

```
int n = 10;  
int nombreAleatori = generadorAleatori.nextInt(n) + 1; // de 1 a  
10  
double nombreAleatori = generadorAleatori.nextDouble(n);
```

Apèndix E. Diagrames de classes amb UML

Sumari

E.1. Especificadors d'accés	964
E.2. Altres especificadors	964
E.3. Relacions entre classes	965
E.4. Miscelània	965

E.1. Especificadors d'accés

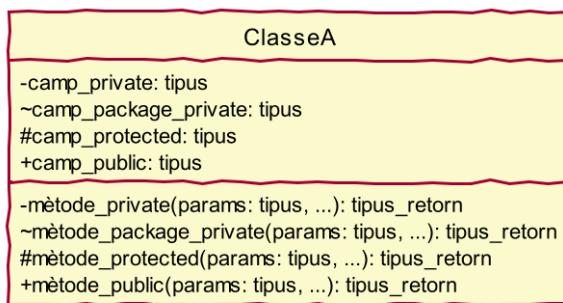


Figura E.1. Classe dibuixada a mà

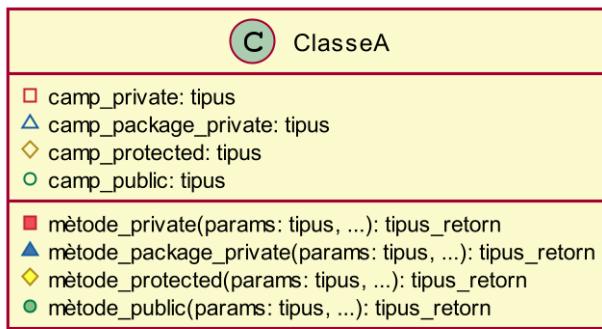
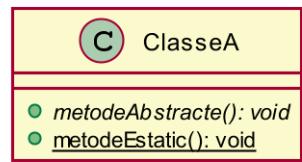
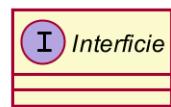
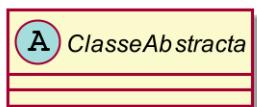


Figura E.2. Símbols utilitzats als apunts

E.2. Altres especificadors



E.3. Relacions entre classes

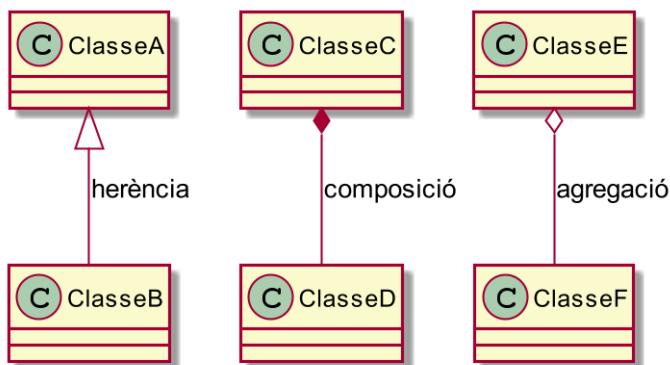


Figura E.3. Relacions entre classes

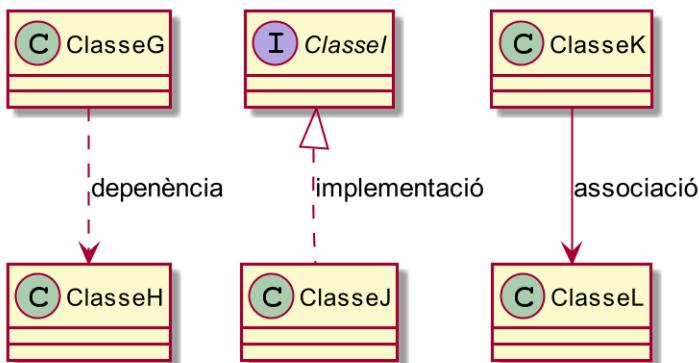


Figura E.4. Relacions entre classes

E.4. Miscelània

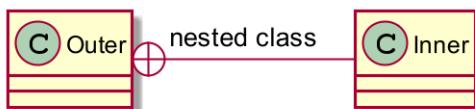


Figura E.5. Relacions entre classes



La relació anterior fa referència a la **implementació** no al **disseny** i no es considera una relació estàndard dins de UML.

S'ha afegit perquè s'utilitza en algun dels diagrames dels apunts.

Apèndix F. Operacions amb bits

Sumari

F.1. Operacions amb bits	968
--------------------------------	-----

F.1. Operacions amb bits

En Java, existeixen quatre operacions amb bits:

- (\sim) negació.
- ($\&$) i binari.
- (\mid) o binari.
- (\wedge) o exclusiu binari.

Quan una operació de bits es realitza sobre un enter la operació es du a terme als bits corresponsants.

Per exemple per calcular $46 \& 13$ primer es convertirien els valors enters a seqüències de bits i després es faria la operació $\&$.

Les operacions bit a bit funcionen de la següent manera:

x	$\sim x$
0	1
1	0

x	y	$x \& y$	$x \mid y$	$x \wedge y$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Vegem el resultat de les operacions anteriors operant amb els valors 54 i 81 (ints).

```

55 = 00000000 00000000 00000000 00110111
81 = 00000000 00000000 00000000 01010001

00000000 00000000 00000000 00110111
& 00000000 00000000 00000000 01010001
-----
00000000 00000000 00000000 00010001

```

```

00000000 00000000 00000000 00110111
| 00000000 00000000 00000000 01010001
-----
00000000 00000000 00000000 01110111

00000000 00000000 00000000 00110111
^ 00000000 00000000 00000000 01010001
-----
00000000 00000000 00000000 01100110

~ 00000000 00000000 00000000 00110111 =
11111111 11111111 11111111 11001000

```

A part de les operacions que treballen sobre els bits individuals existeixen tres operacions de desplaçament de bits, aquestes operacions permeten desplaçar el patró de bits cap la dreta o cap a l'esquerra.

- (`<<`) Desplaçament a l'esquerra
- (`>>`) Desplaçament a la dreta mantenint el bit de signe
- (`>>>`) Desplaçament a la dreta sense mantenir el bit de signe

Vegem com funcionen:

```

1 = 00000000 00000000 00000000 00000001
-2147483648 = 10000000 00000000 00000000 00000000

int a = 1;
a << 5 = 00000000 00000000 00000000 00100000
a << 31 = 10000000 00000000 00000000 00000000
a << 32 = 00000000 00000000 00000000 00000001

int a = -2147483648;
a >> 1 = 11000000 00000000 00000000 00000000
a >> 31 = 11111111 11111111 11111111 11111111
a >> 32 = 10000000 00000000 00000000 00000000

a >>> 1 = 01000000 00000000 00000000 00000000
a >>> 31 = 00000000 00000000 00000000 00000001
a >>> 32 = 00000000 00000000 00000000 00000000

```

Apèndix G. Llicència

Sumari

G.1. Llicència pel text	971
G.2. Llicència pel codi	972

G.1. Llicència pel text

 Aquesta obra està subjecta a una llicència de [Reconeixement-CompartirIgual 4.0 Internacional de Creative Commons](#)¹

Per reutilitzar parts d'aquest projecte heu de seguir les indicacions següents:

Atribució

Heu de proporcionar crèdit als autors incloent un enllaç a la URL original d'aquest projecte, o a una còpia estable i de lliure accés, i que contingui la informació completa de l'autoria tal com apareix aquí.

Compartir igual

Si feu modificacions o afegitons a la pàgina que reutilitzeu, heu de llicenciar-los sota la llicència Reconeixement-CompartirIgual de Creative Commons o posterior.

Indicar els canvis

Si en feu modificacions o afegitons, heu d'indicar que l'obra original ha estat modificada. En una wiki o un repositori de Git la informació històrica que proporciona el sistema és suficient.

Nota de llicència

Cada còpia o versió modificada que redistribuïu ha d'incloure una nota de llicència indicant que l'obra s'ha alliberat sota la CC-BY-SA i un enllaç a la llicència original, o bé el text complet de la llicència.

¹ <http://creativecommons.org/licenses/by-sa/4.0/>

G.2. Llicència pel codi

També podeu optar per utilitzar les parts de codi que hi ha en els vostres propis projectes sota els termes de la llicència GPL de GNU.

Els termes exactes són aquests:

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <<http://www.gnu.org/licenses/>>.

Index

Símbols

//, 32
@FunctionalInterface, 603
@override, 382
=, 52
1>, 65

A

abstracció, 336
abstract, 402
accés directe, 263, 743
accés seqüencial, 263, 263, 743, 743
accessors, 338
agregació, 345
amagament de la informació, 335
ArrayList, 202
assignació de variables, 39

B

Binari, 41
binarySerach, 200
boolean, 45
break, 94, 103
bucle infinit, 98
byte, 44

C

CallableStatement, 898
char, 44
Character, 155
checked exceptions, 448
Collator, 158

composició, 346
constants, 61
constructor per defecte, 316
Constructor per defecte, 343
continue, 103

D

DatabaseMetaData, 881
declarar una variable, 38
default package, 191
Diagrama de classes UML, 333
Diagrama de control de flux, 78
directori de treball, 244, 749
double, 48
do-while, 99
Downcasting, 375

E

encapsulació, 335
Entrada i sortida de fitxers
Scanner, 287, 789
equals, 83, 201
Errors d'execució, 445
EventHandler, 643
excepcions, 445
Exception, 446
exists, 249, 753
exponent, 949
expressions lambda, 607
extends, 371

F

File, 247, 751
FileInputStream, 269, 771

FileOutputStream, 269, 772
fill, 201
final, 61, 394
finally, 453
float, 48
for, 101
for-each, 111
Fully Qualified Name, 190
funció, 162
java.nio, 243, 737
java.security, 205
java.sql, 205
java.swing, 205
java.time, 204
java.util, 200
java.util.ArrayList, 202
javafx, 205
jerarquia de classes, 372

G

getters, 338

H

heap, 36
herència múltiple, 370
Hexadecimal, 41

I

identificador, 33
if, 84
if-else, 85
immutables, 396
implements, 415
import, 195
inner class, 346, 592
instanceof, 376
int, 41
interface, 414
interfície funcional, 603

J

jar, 196
java.io, 205, 243, 737
java.lang, 199
java.net, 205

L

LIFO, 36, 321
lifo, 166
literals, 61
Locale, 158
long, 43

M

mantissa, 949
Matrius, 107
mètodes, 162
mutators, 338

N

null, 318

O

Observer, 665
Octal, 41
operador diamant, 476
Operadors,
overflow, 946
overloading, 387
overriding, 383

P

paradigma de programació, 73
parseDouble, 155
parseInt, 155
pas per valor, 168
polimorfisme, 419
Príncipi de responsabilitat única, 421
príncipi de responsabilitat única, 654
Príncipi d'obert-tancat, 422
príncipi d'obert-tancat, 655
printf, 67
problema del diamant, 375
propietats, 338
Pseudocodi, 77

R

Random, 960, 960
RandomAccessFile, 803
 getFilePointer, 806
 length, 806
 seek, 806
recoln·lector de brossa, 37
Recursivitat,
reflection, 410
registre d'activació, 166
ResultSetMetaData, 885
return, 163
ruta, 245, 750
ruta absoluta, 245, 750
 ruta canònica, 252, 756
 ruta relativa, 245, 750

S

Scanner, 69
SelectionSort, 115
Seqüències d'escapament, 24

setters, 338
shadowing, 183
short, 44
signatura del mètode, 163
signatura d'un mètode, 327
sort, 200
stack, 36
stack frame, 166
StackOverflowError, 214
static nested class, 592
String, 139
 mètodes, 145
StringBuffer, 159
StringBuilder, 159
super, 382, 386
switch, 92
System.out.print(), 23
System.out.println(), 23

T

teorema de la programació estructurada, 73
this, 331
throw, 446
tipus de dades, 35
tipus de dades per referència, 37, 322
tipus de dades primitius, 37, 322
TODO, 189, 393, 393, 415, 415, 415, 689
toString, 202
try/catch, 450

U

UML, 333
uml
 agregació, 345

associació, 345
classe, 334
composició, 346
dependència, 344
underflow, 946
Upcasting, 375

V

valueOf, 155
variable, 33
variables estàtiques, 317
variables locals, 181
vector, 119

W

while, 97