

# Manual Java I

## Aprende a programar con Java desde 0

José Ángel Navarro  
V 1.4 (2023)



Esta obra está bajo una licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 4.0 Internacional

## Índice

1.	Introducción .....	12
2.	Lenguajes de programación .....	13
2.1	Tipos .....	13
2.2	Programación estructurada .....	16
3.	Lo primero que tenemos que aprender.....	18
4.	¿Qué es Java? .....	19
5.	Preparación de nuestro entorno de desarrollo.....	20
5.1	Instalando Java .....	20
5.1.1	JDK.....	20
5.1.2	Geany .....	26
6.	Comenzando .....	30
7.	Conceptos iniciales.....	36
7.1	Variables.....	39
7.2	Constantes.....	43
7.3	Operaciones .....	44
7.4	Operadores de asignación.....	46
7.5	Operadores unarios aritméticos .....	47
7.6	Prioridad de operadores .....	47
7.7	Casting.....	48
7.8	Ejercicios.....	49
8.	Estructuras de control.....	50
8.1	Estructuras condicionales .....	50
8.1.1	if.....	50
8.1.2	switch .....	60
8.2	Estructuras de repetición .....	63
8.2.1	while .....	63
8.2.2	do.....	65
8.2.3	for .....	66
8.2.4	¿Cuál debo de usar? .....	67
8.2.5	Break y continue.....	69
8.3	Ejercicios.....	70
9.	API .....	71
10.	Entrada por teclado.....	74
10.1	Ejercicios.....	79
11.	Vectores .....	80

11.1	Ordenación.....	85
11.2	Matrices.....	86
11.3	Ejercicios.....	90
12.	Métodos .....	91
12.1	Static.....	92
12.2	Modificador de ámbito .....	92
12.3	Return.....	93
12.4	Paso de parámetros .....	95
12.5	Recursividad .....	100
12.6	Ejercicios.....	102
13.	Clases.....	103
13.1	Clases vs objetos .....	104
13.2	¿Qué pasos sigue java para crear un objeto? .....	106
13.3	Constructores .....	106
13.4	La referencia this.....	109
13.5	Modificadores de acceso.....	110
13.6	Static.....	113
13.7	Final .....	114
13.8	Vectores de objetos .....	115
14.	Herencia .....	116
14.1	Sobreescritura .....	122
14.2	Sobrecarga.....	123
14.3	toString.....	124
14.4	Clases abstractas .....	124
14.5	Polimorfismo .....	126
14.6	Ejercicios.....	130
15.	Object .....	132
15.1	toString() .....	132
15.2	equals().....	134
15.3	Asignación de Objetos.....	137
15.4	Vectores de Objetos.....	139
15.5	Constructor copia.....	139
15.6	Wrappers.....	140
16.	Interfaces.....	142
16.1	Declaración de una interfaz .....	142
16.2	Implementación de una interfaz en una clase .....	143

16.3	Jerarquía entre interfaces .....	143
16.4	Interfaces como tipos.....	143
16.5	Cambios en JDK8 .....	145
16.5.1	Default.....	145
16.5.2	Interface funcional .....	146
16.5.3	Expresiones Lambda.....	146
16.6	¿Qué es Comparable? .....	148
16.6.1	¿Para qué sirve Comparable?.....	148
16.6.2	¿Cómo funciona compareTo? .....	148
16.6.3	¿Cómo comparar dos objetos? .....	149
16.6.4	¿Cómo llamar a compareTo? .....	149
16.6.5	Ejercicio .....	150
17.	La clase String.....	151
18.	La clase Math.....	154
19.	Anexo I. ArrayList .....	156
19.1	Estructuras para recorrer colecciones .....	159
20.	Anexo II. Excepciones .....	163
21.	Anexo III. Ordenar objetos de una colección cuando hay nulos .....	166
22.	Anexo IV. Algebra de Boole.....	169
23.	Bibliografía .....	171
24.	Ejercicios.....	172

## Índice de imágenes

Ilustración 1. Ejemplo código máquina .....	13
Ilustración 2. Ejemplo ensamblador.....	13
Ilustración 3. Ejemplo Python .....	14
Ilustración 4. Compilador C.....	15
Ilustración 5. Interprete Python.....	15
Ilustración 6. Compilador Java .....	16
Ilustración 7. Instalador JDK.....	21
Ilustración 8.Instalación JDK.....	22
Ilustración 9.Instalación JDK 2.....	22
Ilustración 10. Fin instalación JDK .....	22
Ilustración 11. Carpeta BIN .....	23
Ilustración 12. Propiedades equipo .....	23
Ilustración 13. Configuración avanzada .....	24
Ilustración 14. Variables de entorno .....	24
Ilustración 15. Variables de entorno 2 .....	25
Ilustración 16. Añadir variable de entorno .....	25
Ilustración 17. Classpath .....	26
Ilustración 18. Classpath 2 .....	26
Ilustración 19. Instalación de Geany .....	27
Ilustración 20. Instalación de Geany 2 .....	27
Ilustración 21. Icono de Geany.....	27
Ilustración 22. Entorno del IDE.....	27
Ilustración 23. Crear una nueva clase .....	28
Ilustración 24. Comandos de construcción .....	28
Ilustración 25. Configuración de los comandos .....	29
Ilustración 26. Creación de un nuevo programa.....	31
Ilustración 27. Código predefinido.....	31
Ilustración 28. Botones de Geany .....	32
Ilustración 29. Compilación.....	32
Ilustración 30. Resultado de ejecución .....	33
Ilustración 31. Comentarios .....	33
Ilustración 32. Salida comentarios .....	33
Ilustración 33. Comentarios 2 .....	34
Ilustración 34. Bloque de código .....	34
Ilustración 35. Punto y coma.....	35
Ilustración 36. Primer ejemplo .....	36
Ilustración 37. Tipos de datos .....	36
Ilustración 38. Tabla ASCII.....	37
Ilustración 39. Nueva línea.....	38
Ilustración 40. Ejemplo nueva línea .....	38
Ilustración 41. Ejemplo rango de datos.....	39
Ilustración 42. Ejemplo rango de datos 2.....	39
Ilustración 43. Variables no inicializadas .....	40
Ilustración 44. Variables no inicializadas 2 .....	40
Ilustración 45. Variables inicializadas 2 .....	41
Ilustración 46. Variables inicializadas .....	41

Ilustración 47. Tipos de variable 2.....	42
Ilustración 48. Tipos de variable .....	42
Ilustración 49. Constantes.....	43
Ilustración 50. Operadores aritméticos .....	44
Ilustración 51. Salida operadores.....	44
Ilustración 52. División decimal.....	45
Ilustración 53. Resultado división decimal.....	45
Ilustración 54. Concatenación .....	45
Ilustración 55. Ejemplo concatenado.....	46
Ilustración 56. Suma con asignación .....	46
Ilustración 57. Suma sin asignación .....	46
Ilustración 58. Post incremento .....	47
Ilustración 59.Pre incremento.....	47
Ilustración 60. Resultado Pre incremento.....	47
Ilustración 61. Resultado Post incremento .....	47
Ilustración 62. Casting implícito .....	48
Ilustración 63. Casting explícito .....	49
Ilustración 64.Estructura del if .....	50
Ilustración 65.Primer if.....	51
Ilustración 66. Ejecución primer if .....	51
Ilustración 67Ejecución del primer if 2.....	51
Ilustración 68. Segundo if.....	51
Ilustración 69. Ejecución segundo if.....	52
Ilustración 70. Tercer if .....	52
Ilustración 71. Ejecución tercer if.....	53
Ilustración 72. Estructura completa del if .....	53
Ilustración 73. Cuarto if.....	53
Ilustración 74. Quinto if.....	54
Ilustración 75. If anidados .....	54
Ilustración 76. Resultado if anidados 1 .....	54
Ilustración 77. Resultado if anidados 2 .....	55
Ilustración 78. Resultado if anidados 3 .....	55
Ilustración 79. Ejemplo sin identación .....	55
Ilustración 80. Operadores lógicos.....	57
Ilustración 81. Salida operadores lógicos.....	57
Ilustración 82. Condiciones cortas .....	58
Ilustración 83. Operador ternario .....	59
Ilustración 84. Estructura switch.....	60
Ilustración 85. switch aprox1 .....	60
Ilustración 86. Salida switch aprox1.....	60
Ilustración 87. switch aprox2 .....	61
Ilustración 88. Salida switch aprox2.....	61
Ilustración 89.switch aprox3 .....	61
Ilustración 90. Salida switch aprox3.....	61
Ilustración 91. switch aprox4 .....	62
Ilustración 92. Salida switch aprox4.....	62
Ilustración 93. switch aprox5 .....	62
Ilustración 94. Estructura while .....	63

Ilustración 95. while aprox 1 .....	63
Ilustración 96. Salida while aprox 1.....	64
Ilustración 97. while aprox 2 .....	64
Ilustración 98. Salida while aprox 2.....	64
Ilustración 99. Estructura do .....	65
Ilustración 100. do.....	65
Ilustración 101. Estructura for .....	66
Ilustración 102. for .....	66
Ilustración 103. for inverso .....	67
Ilustración 104. Condición con char .....	68
Ilustración 105. Salida condición char.....	68
Ilustración 106. Break.....	69
Ilustración 107. API 1.....	71
Ilustración 108. API 2.....	71
Ilustración 109. API 3.....	72
Ilustración 110. API 4.....	72
Ilustración 111. API 5.....	72
Ilustración 112. API 6.....	73
Ilustración 113. Prueba Scanner .....	75
Ilustración 114. Salida Scanner .....	75
Ilustración 115. nextDouble().....	76
Ilustración 116. Salida nextDouble() .....	76
Ilustración 117. Salida excepción .....	76
Ilustración 118. String .....	77
Ilustración 119. Salida String .....	78
Ilustración 120. For String .....	78
Ilustración 121. Salida for String .....	78
Ilustración 122. Salida 2 for string.....	78
Ilustración 123. Vector 1 .....	81
Ilustración 124. Salida vector 1 .....	81
Ilustración 125. Vector 2 .....	82
Ilustración 126. Vector 3 .....	83
Ilustración 127. Salida vector 3 .....	84
Ilustración 128. Vector 4 .....	84
Ilustración 129. Salida vector 4 .....	84
Ilustración 130. Random .....	85
Ilustración 131. Algoritmo de la burbuja .....	85
Ilustración 132. Burbuja 1 .....	86
Ilustración 133. Salida burbuja.....	86
Ilustración 134. Burbuja 2 .....	86
Ilustración 135. Matriz 1 .....	87
Ilustración 136. Salida Matriz 1 .....	88
Ilustración 137. Matriz 2 .....	88
Ilustración 138. Salida matriz 2 .....	88
Ilustración 139. Matriz 3 .....	89
Ilustración 140. Salida matriz 3 .....	90
Ilustración 141. Ejemplo void.....	94
Ilustración 142. Salida void .....	94

Ilustración 143. Return.....	94
Ilustración 144. Salida Return .....	95
Ilustración 145. Parámetros formales y reales.....	95
Ilustración 146. Parámetros por valor.....	96
Ilustración 147. Salida por valor.....	96
Ilustración 148. Parámetros por referencia .....	98
Ilustración 149. Salida Parámetros por referencia.....	98
Ilustración 150. Recursividad .....	100
Ilustración 151. Factorial iterativo .....	101
Ilustración 152. Primera aproximación clases.....	104
Ilustración 153. Test Clases 1 .....	105
Ilustración 154. Salida Clases 1 .....	105
Ilustración 155. Segunda aproximación clases.....	107
Ilustración 156. Test Clases 2 .....	107
Ilustración 157. Salida clases 2 .....	107
Ilustración 158. Tercera aproximación.....	108
Ilustración 159. Test Clases 3 .....	108
Ilustración 160. Constructor sin this .....	109
Ilustración 161. Constructor sin this, mal .....	109
Ilustración 162. Constructor con this .....	110
Ilustración 163. Cuarta aproximación .....	111
Ilustración 164. Acceso erróneo.....	112
Ilustración 165. Setter's y getter's.....	113
Ilustración 166. Propiedad static.....	114
Ilustración 167. Modificador final .....	115
Ilustración 168. Clase coche .....	115
Ilustración 169. main.....	116
Ilustración 170. Salida null .....	116
Ilustración 171. Diagrama herencia .....	117
Ilustración 172. Factor común en herencia.....	117
Ilustración 173. Clases padre e hijas .....	118
Ilustración 174. Ejemplo clase padre .....	119
Ilustración 175. Clase hija 1.....	120
Ilustración 176. Clase hija 2.....	121
Ilustración 177. Clase hija 3.....	121
Ilustración 178. Ejemplo de main con herencia .....	121
Ilustración 179. Diagrama animales .....	122
Ilustración 180. Sobreescritura .....	123
Ilustración 181. Sobrecarga.....	123
Ilustración 182. Sobrecarga 2 .....	124
Ilustración 183. Clase abstracta .....	125
Ilustración 184. Clase abstracta 2 .....	126
Ilustración 185. Producto .....	127
Ilustración 186. Trigo.....	127
Ilustración 187. Leche .....	127
Ilustración 188. Test Polimorfismo .....	128
Ilustración 189. Salida Polimorfismo.....	128
Ilustración 190. Trigo 2 .....	129

Ilustración 191. Leche 2 .....	129
Ilustración 192. Error Polimorfismo .....	129
Ilustración 193. Error Polimorfismo 2 .....	130
Ilustración 194. instanceof .....	130
Ilustración 195. toString .....	132
Ilustración 196. Ejemplo toString .....	132
Ilustración 197. toString correcto .....	133
Ilustración 198. equals 1 .....	134
Ilustración 199. equals 2 .....	134
Ilustración 200. Salida equals 2 .....	134
Ilustración 201. Asignación a Objetos .....	137
Ilustración 202. Asignación a Objetos 2 .....	137
Ilustración 203. Salida Asignación a Objetos .....	138
Ilustración 204. Constructor copia .....	139
Ilustración 205. Main constructor copia .....	140
Ilustración 206. Salida constructor copia .....	140
Ilustración 207. wrapper's .....	141
Ilustración 208. interface .....	142
Ilustración 209. implements .....	143
Ilustración 210. Herencia e implementación .....	143
Ilustración 211. Interfaces como tipos 1 .....	143
Ilustración 212. Interfaces como tipos 2 .....	144
Ilustración 213. Interfaces como tipos 3 .....	144
Ilustración 214. Interfaces como tipos 4 .....	144
Ilustración 215. Salida Interfaces como tipos .....	144
Ilustración 216. Métodos por defecto 1 .....	145
Ilustración 217. Métodos por defecto 2 .....	145
Ilustración 218. Métodos por defecto 3 .....	145
Ilustración 219. Salida Métodos por defecto .....	145
Ilustración 220. Métodos por defecto 4 .....	146
Ilustración 221. Salida Métodos por defecto 4 .....	146
Ilustración 222. Expresiones Lambda 1 .....	147
Ilustración 223. Expresiones Lambda 2 .....	147
Ilustración 224. Salida Expresiones Lambda 2 .....	147
Ilustración 225. Expresiones Lambda 3 .....	147
Ilustración 226. Expresiones Lambda 4 .....	147
Ilustración 227. Salida Expresiones Lambda 4 .....	148
Ilustración 228. Expresiones Lambda 5 .....	148
Ilustración 229. Expresiones Lambda 6 .....	148
Ilustración 230. Salida Expresiones Lambda 6 .....	148
Ilustración 231. compareTo .....	149
Ilustración 232. Prueba compareTo .....	150
Ilustración 233. Extracto de métodos Math .....	154
Ilustración 234. Ejemplo Math .....	155
Ilustración 235. Salida Math .....	155
Ilustración 236. Ejemplo ArrayList 1 .....	156
Ilustración 237. Ejemplo ArrayList 2 .....	158
Ilustración 238. Salida Ejemplo ArrayList 2 .....	158

Ilustración 239. Ejemplo ArrayList 3.....	158
Ilustración 240. Salida Ejemplo ArrayList 3 .....	158
Ilustración 241. Ejemplo ArrayList 4.....	159
Ilustración 242. Salida Ejemplo ArrayList 4 .....	159
Ilustración 243. Ejemplo for each .....	159
Ilustración 244. Salida Ejemplo for each.....	160
Ilustración 245. Ejemplo de iterador.....	161
Ilustración 246. Salida Ejemplo de iterador .....	161
Ilustración 247. Ejemplo ArraList 5 .....	161
Ilustración 248. Test Ejemplo ArraList 5.....	162
Ilustración 249. Salida Ejemplo ArraList 5 .....	162
Ilustración 250. wrappers.....	162
Ilustración 251. Ejemplo de excepción .....	163
Ilustración 252. Jerarquía de excepciones .....	163
Ilustración 253. Excepciones 1 .....	164
Ilustración 254. Salida Excepciones 1.....	164
Ilustración 255. Excepciones 2 .....	164
Ilustración 256. Salida excepciones 2.....	164
Ilustración 257. Excepciones 3 .....	165
Ilustración 258. Comparator 1 .....	166
Ilustración 259. Comparator 2 .....	166
Ilustración 260. Comparator salida 1 .....	167
Ilustración 261. Comparator 3 .....	167
Ilustración 262. Salida 2 Comparator .....	167
Ilustración 263. Comparator 4 .....	168
Ilustración 264. Comparator 5 .....	168
Ilustración 265. Salida 3 Comparator .....	168

## Índice de tablas

Tabla 1. Variables, memoria 1.....	41
Tabla 2. Variables, memoria 2 .....	41
Tabla 3. Variables, memoria 3.....	41
Tabla 4. Variables, memoria 4.....	42
Tabla 5. Ejemplos de tipos letra .....	42
Tabla 6. Operadores aritméticos.....	44
Tabla 7. Operadores de asignación .....	46
Tabla 8. Operadores de incremento .....	47
Tabla 9 Prioridad de operadores.....	48
Tabla 10. Operadores de comparación .....	52
Tabla 11. Operadores lógicos.....	56
Tabla 12. Tabla de verdad AND y OR.....	56
Tabla 13.Tabla NOT .....	56
Tabla 14. Tabla XOR.....	57
Tabla 15.Ejemplo parámetros por valor 1.....	96
Tabla 16. Ejemplo parámetros por valor 2.....	97
Tabla 17. Ejemplo parámetros por valor 3.....	97
Tabla 18. Ejemplo parámetros por valor 4.....	97
Tabla 19. Ejemplo parámetros por valor 5.....	97

Tabla 20. Ejemplo parámetros por valor 6.....	98
Tabla 21. Ejemplo parámetros por referencia 1 .....	99
Tabla 22. Ejemplo parámetros por referencia 2 .....	99
Tabla 23. Ejemplo parámetros por referencia 3 .....	99
Tabla 24. Ejemplo parámetros por referencia 4 .....	99
Tabla 25. Modificadores de acceso .....	111
Tabla 26. Clase Math .....	154
Tabla 27. Métodos ArrayList .....	157

## 1. Introducción

Este manual ha sido realizado en base a la experiencia en la explicación de la asignatura de programación en CGFS a alumnos que no han programado nunca y se encuentran con un lenguaje orientado a objetos. No trato de que los alumnos no atiendan las explicaciones de sus profesores, sino de darles una base aclaratoria desde 0.

El manual está basado en una serie de aproximaciones hasta llegar a comprender la POO. Para ello está dividido en tres grandes partes:

1. Programar funcionalmente desde un punto de vista algorítmico. Viendo lo indispensable para comenzar a resolver problemas.
2. Añadir métodos y clases, acercándonos ya a la POO.
3. Explicar la herencia, abstracción y demás conceptos de la POO.

Obviamente al explicar un lenguaje orientado a objetos nos encontramos con que hay cosas que hay que explicar pero que no es el momento por la dificultad que comportan. Sólo cuando se ha visto todo se cierra el círculo y tenemos una visión completa.

He usado aproximaciones hasta llegar a la solución final de forma que el alumno vea todo lo que sucede, de ahí la cantidad de ejemplos.

Mi misión es enseñar a programar no a ser copistas de código o codificadores. En ningún momento mi función es enseñar el uso de una API. Durante toda mi experiencia como profesor me he encontrado gente, con titulación, que no sabe lo que es un vector, sólo conocen algo llamado ArrayList. Me parece una aberración, antes de ir en moto hay que aprender a ir en bicicleta, sólo así podemos comprender el funcionamiento de las cosas para sacarles el mayor partido. Obviamente uso y explico cosas de la API, pero luego, este es un manual inicial, habrá más de mayor dificultad.

Solo aprenderéis a programar haciendo ejercicios, primero en papel y boli. Si no lo hacéis así se aprende por prueba y error (hasta que compile) por tanto no sabéis programar, sabéis probar cosas hasta dar con una que funcione. Debéis de poder leer un código en papel y decir si va a funcionar o no.

## 2. Lenguajes de programación

### 2.1 Tipos

Un lenguaje de programación es un lenguaje que puede ser utilizado para controlar el comportamiento de una máquina, particularmente una computadora. Permite especificar de manera precisa sobre qué datos se tiene que operar, cómo deben ser estos almacenados, transmitidos y qué acciones se debe tomar bajo una variada gama de circunstancias.

Según la dependencia de la máquina en la que se ejecutan los podemos caracterizar por:

- **Lenguajes de bajo nivel**

- Fuerte dependencia de la máquina.
- Muy alejados de nuestro modo de razonar.
- Difíciles de aprender, entender y manipular.

Lenguaje Maquina: Código binario, juego de instrucciones, modos de direccionamiento, operandos ...

20	65	73	74	65	20	65	73-20	75	6E	20	70	72	6F	67
72	61	6D	61	20	68	65	63-68	6F	20	65	6E	20	61	73
73	65	6D	62	6C	65	72	20-70	61	72	61	20	6C	61	20
57	69	6B	69	70	65	64	69-61	24						

Ilustración 1. Ejemplo código máquina

Lenguaje ensamblador: Usa códigos mnemotécnicos haciendo más fácil de leer los programas. Siguen siendo difíciles y dependientes de la máquina.

MOV	DX,010B
MOV	AH,09
INT	21
MOV	AH,00
INT	21

Ilustración 2. Ejemplo ensamblador

Se puede ver como no son fáciles de usar.

- **Lenguajes de alto nivel**

- Mayor independencia de la máquina.
- Más cercanos al lenguaje humano.
- Fortran, Lisp, Algol, Cobol, Basic, Pascal, C.

## Ej. Python

```
numero=int(raw_input("Introduce un numero: "))
if numero>0:
    print "El numero es positivo"
if numero<0:
    print "El numero es negativo"
```

*Ilustración 3. Ejemplo Python*

En este ejemplo, sin tener conocimientos de python igualmente podemos entender que se pide un número y que dependiendo de si es mayor o menor de 0 se muestra un mensaje distinto.

Dentro de los lenguajes de alto nivel también nos podemos encontrar con varios tipos según el paradigma que usen: procedimentales, declarativos, funcionales...

Este manual se centra en Java que es un lenguaje orientado a objetos (en adelante POO). Estos lenguajes estrechan la relación entre código y datos. Ofrecen un mayor nivel de abstracción. Desde que comenzaron a salir lenguajes con POO se han ido añadiendo distintas características, pero hay tres que, bajo mi punto de vista, hacen que se le pueda llamar POO o no. Estas son:

- **Herencia:** Capacidad de establecer relaciones jerárquicas entre objetos.
- **Polimorfismo:** Una operación puede adoptar diferentes formas.
- **Encapsulación:** Damos a conocer solo aquellos detalles de los objetos que son pertinentes, ocultando como se han implementado.

Independientemente del tipo de lenguaje según las características vistas, estos pueden ser interpretados o compilados, dependiendo del proceso de traducción que se realice. La CPU solo entiende código máquina, por lo que hay que traducir el programa fuente, programado en uno de los lenguajes existentes, en código máquina para que pueda ser ejecutado.

- **Intérpretes:** traducen instrucción a instrucción conforme lo necesitan.
- **Compiladores:** traducen todo el programa creando un programa objeto en lenguaje máquina.

Cada uno tiene sus pros y sus contras. Un lenguaje compilado traduce una sola vez todo el código y son más rápidos en ejecución. Los interpretados si una instrucción se repite varias veces la interpretan cada vez ya que va traduciendo línea a línea mientras se ejecuta. Por el contrario son más fáciles de depurar. Hoy en día, dada la capacidad de computo de los ordenadores, no es tan importante la velocidad de ejecución, de hecho cada vez surgen más lenguajes interpretados o mixtos como es Java.

Ejemplo de lenguaje compilado: C

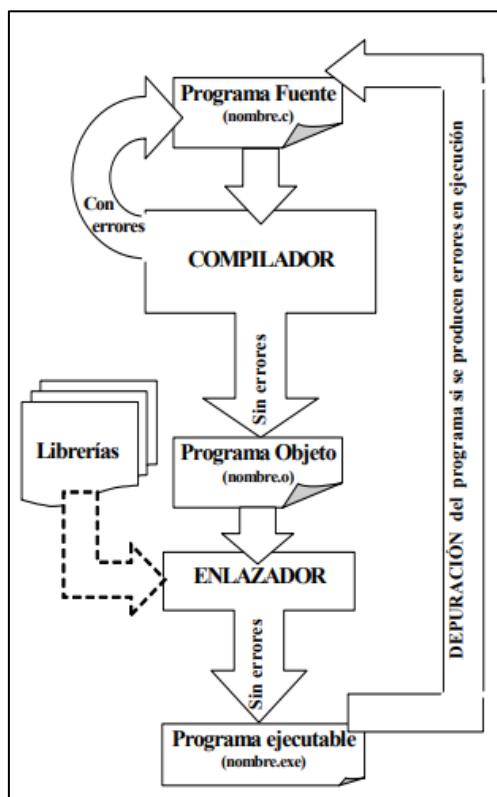


Ilustración 4. Compilador C

Ejemplo de lenguaje interpretado: Python

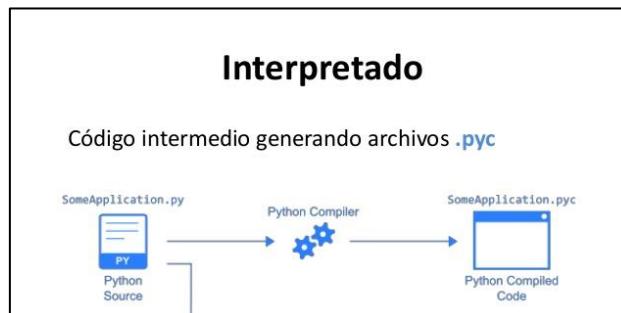


Ilustración 5. Interprete Python

**Java:** Java es un lenguaje que compila generando un código intermedio (bytecode) que se ejecuta en un entorno interpretado llamado Máquina Virtual Java. Un programa realizado en Java se puede ejecutar en cualquier ordenador tenga el sistema operativo que tenga siempre que este instalada la máquina virtual.

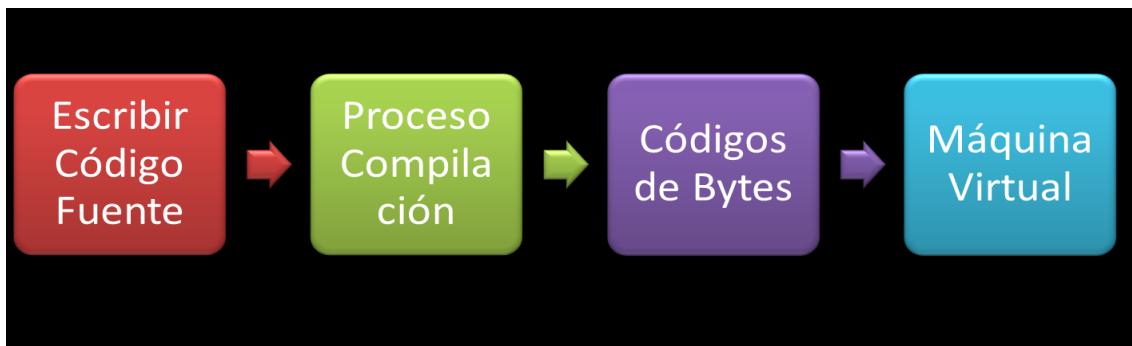


Ilustración 6. Compilador Java

## 2.2 Programación estructurada

A lo largo de la historia de la programación ha habido distintas técnicas de programación: convencional, modular... En este manual sólo voy a centrarme en la programación estructurada que, independientemente de que tengamos módulos en nuestro programa, es la que debemos de usar.

Esta técnica establece que un programa bien estructurado debe cumplir las siguientes condiciones:

- Deberá contener en su código información suficiente para ser comprendido sin necesidad de información adicional.
- Las distintas partes de las que consta un programa deberán ser modificadas o cambiadas sin que esto afecte al resto del programa.
- El programa podrá ser desarrollado por partes fácilmente ensamblables.

### Elementos de la programación estructurada

- **Razonamiento deductivo:** resolución de un problema por medio de pasos consecutivos, donde la salida de un paso será la entrada de otro. Refinamiento. Divide y vencerás
- **Recursos abstractos:** transformar los recursos abstractos en recursos concretos, las ideas en instrucciones.
- **Estructuras básicas:** un programa puede desarrollarse usando solamente tres estructuras básicas:
  - **Estructura secuencial:** secuencia de acciones que se ejecutan una detrás de otra.
  - **Estructura alternativa:** permite que se ejecuten una serie de instrucciones dependiendo de si se cumple o no una determinada condición.

- **Estructura repetitiva:** permite la ejecución repetida de un grupo de instrucciones un número determinado de veces o hasta que se cumpla una condición.

#### Ventajas de la programación estructurada

- Ahorro de tiempo en la labor de codificación.
- Facilidad de localización y corrección de errores.
- Programas sencillos y rápidos.
- Programas fáciles de leer y entender.
- Programas bien documentados.

### 3. Lo primero que tenemos que aprender

- **No hay ningún lenguaje mejor que otro:** No perdáis el tiempo en guerras inútiles entre lenguajes. Hay lenguajes más nuevos o más populares, pero no mejores. Cada lenguaje es adecuado para un contexto o tipo de aplicación determinado. No hay ninguno que sea mejor que todos los demás de forma absoluta. No dejéis que el marketing os engañe.
- **Tú no eres tu código:** Tendemos a relacionar nuestra valía personal con la calidad del código que escribimos. Y eso es un error. Si mi autoestima dependiera de lo bueno que es el código que yo escribo, hace tiempo que me hubiera tirado a la vía del tren. Esto se aplica también a la inversa. No vayas tan rápido en evaluar la calidad de un programador mirando su código. No sabes en qué condiciones tuvo que escribirlo (a lo mejor parte lo heredó de otro, el tiempo asignado...)
- **Más no es mejor:** No seas una máquina de “vomitar” código. Sé una máquina de solucionar problemas. Y si es con poco código aún mejor.
- **Leerás más código del que nunca vas a programar:** Pocas veces tendrás la oportunidad de programar algo de cero. Incluso en esos casos dependerás de librerías y servicios de terceros.
- El 99% de los tiempos estarás **colaborando** en un proyecto ya en marcha (o depurando uno viejo...).
- **Programar es mucho más que escribir código:** Programar es una actividad social. Eso del programador encerrado en su cubículo es un mito. Programar requiere entender bien los requisitos de lo que vas a programar, el contexto en que se va a utilizar el programa, el perfil del usuario, ... Para ser un buen programador tienes que desarrollar no sólo código sino también habilidades sociales.
- **Nunca escribirás un código perfecto:** El código perfecto no existe. Nadie ha creado nunca un código perfecto, no vas a ser tú el primero. Intentarlo te puede llevar al grave problema de la optimización prematura donde, en la búsqueda de la perfección, acabas escribiendo un montón de código inútil. Inútil porque intenta prever posibles problemas (ej. de escalabilidad) que a día de hoy no existen y que puede que nunca lleguen. Una cosa es definir un código bien legible y fácil de extender y la otra perder el tiempo preparando el código en previsión de una guerra nuclear.
- **Hay muchas formas de solucionar un problema e infinitas de no hacerlo.**

## 4. ¿Qué es Java?

Java es un lenguaje orientado a objetos de propósito general. Aunque Java comenzara a ser conocido como un lenguaje de programación de applets que se ejecutan en el entorno de un navegador web, hoy en día se puede utilizar para construir cualquier tipo de proyecto.

Su sintaxis es muy parecida a la de C y C++ pero hasta ahí llega el parecido. Java no es una evolución ni de C++ ni un C++ mejorado.

Los padres de Java son James Gosling (emacs) y Bill Joy (Sun). Java desciende de un lenguaje llamado Oak.

Los criterios de diseño de Java fueron:

- Independiente de la máquina.
- Seguro para trabajar en red.
- Potente para sustituir código nativo.

En el diseño de Java se prestó especial atención a la seguridad. Existen varios niveles de seguridad en Java, desde el ámbito del programador, hasta el ámbito de la ejecución en la máquina virtual.

Java realiza comprobación estricta de tipos durante la compilación (fuertemente tipado), evitando con ello problemas tales como el desbordamiento de la pila. Pero, es durante la ejecución donde se encuentra el método adecuado según el tipo de la clase receptora del mensaje.

Todas las instancias de una clase se crean con el operador *new*, de manera que un recolector de basura se encarga de liberar la memoria ocupada por los objetos que ya no están referenciados. La máquina virtual de Java gestiona la memoria dinámicamente.

Java también posee mecanismos para garantizar la seguridad durante la ejecución comprobando, antes de ejecutar código, que este no viola ninguna restricción de seguridad del sistema donde se va a ejecutar.

Otra característica de Java es que está preparado para la programación concurrente sin necesidad de utilizar ningún tipo de biblioteca.

Finalmente, Java posee un gestor de seguridad con el que poder restringir el acceso a los recursos del sistema.

## 5. Preparación de nuestro entorno de desarrollo

Hoy en día los programadores usan entornos de desarrollo (en adelante IDE) que les ayudan en su tarea. Entre las características de estos entornos podemos encontrar:

- Editor de texto con colores.
- Compilador.
- Intérprete.
- Herramientas de automatización.
- Depurador.
- Posibilidad de ofrecer un sistema de control de versiones.
- Factibilidad para ayudar en la construcción de interfaces gráficas de usuario.
- Refactorización.
- Autocomplementado de código.

Entre los IDE más usados para programar con Java están: Netbeans, Eclipse, IntelliJ...

El problema de usar uno de estos IDE para aprender y dar los primeros pasos es que hacen muchas cosas solos, nosotros conseguimos que nuestros programas funcionen pero no sabemos cómo ni porqué ya que ha sido el IDE el que nos ha corregido errores o nos ha dicho como escribirlo. Por ello nosotros vamos a usar otro IDE más liviano pero muy conocido, compatible con Java, C, C++, PHP, Python... que es Geany. A Geany se le pueden añadir plugins para ayudarnos en todo lo comentado anteriormente, cosa que no haremos.

### 5.1 Instalando Java

#### 5.1.1 JDK

Para poder programar en java debemos de instalar el JDK (Java Development Kit). El JDK contiene herramientas de consola y herramientas de compilación que pueden ser usadas por un IDE. El JDK incluye también el JRE (Java Runtime Environment) que contienen las librerías de clase y la MVJ. Entre los comandos que contiene el JDK podemos citar:

- `java`: La máquina virtual.
- `javac`: El compilador.
- `javadoc`: generador de documentación.

Para instalar el JDK lo primero es bajárnoslo de la web de Oracle:  
<https://www.oracle.com/java/technologies/downloads/>



Con Java 9, Oracle se comprometió a lanzar una nueva versión de Java cada 6 meses, y de momento lo está cumpliendo. Con el lanzamiento de Java 11 hay cambio de licencia y el soporte a largo plazo (LTS) asegurado para el JDK. Por tanto, los JDK no son gratuitos. El JDK 11 inicia una nueva era en la licencia de uso. Hasta ahora podías descargar y programar con el Kit de Desarrollo de Java oficial de Oracle y luego poner tu aplicación en producción o distribuirla sin tener que pagar nada al gigante del software. Sin embargo, a partir de Java 11 y del JDK 11, aunque puedes seguir desarrollando con él, tendrás que pagar una licencia a Oracle si quieras utilizarlo para poner las aplicaciones en producción. El coste es de 2,5 dólares al mes por cada usuario de escritorio, y de 25 dólares por procesador en el caso de aplicaciones de servidor. Esto no afecta a versiones anteriores del JDK, por lo que, si usas Java 8, 9 o 10 sigue siendo gratuito.

Sin embargo, sabiendo los cambios de licencias que estaban preparando, Oracle ha trabajado para hacer que el OpenJDK se haya equiparado en todos los aspectos al JDK, hasta el punto de que se puede decir que el OpenJDK y el JDK son idénticos desde un punto de vista técnico, desde la versión 11. Por tanto podemos usar OpenJDK.

Nosotros vamos a usar el JDK 8 que es más que suficiente para aprender a programar y para nuestros intereses, Es una versión LTS y que además sigue siendo gratuita.

La podéis descargar en este enlace:

<https://www.oracle.com/es/java/technologies/javase/javase8-archive-downloads.html>

Es importante descargar la adecuada para nuestro sistema operativo (32 o 64 bits). Se nos pedirá que nos registremos, no hay problema es gratuito y podemos desactivar las notificaciones.

Una vez descargado el JDK procedamos a su instalación. Lo ejecutamos y seguimos las instrucciones que nos van saliendo en pantalla:

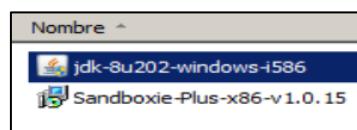


Ilustración 7. Instalador JDK

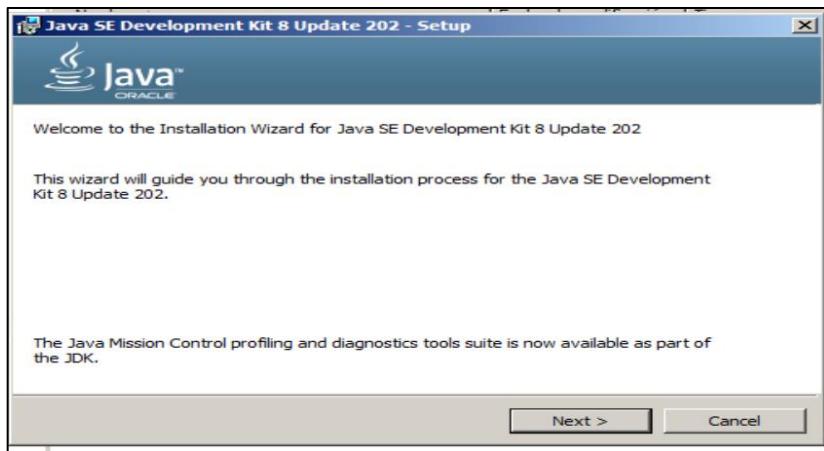


Ilustración 8. Instalación JDK

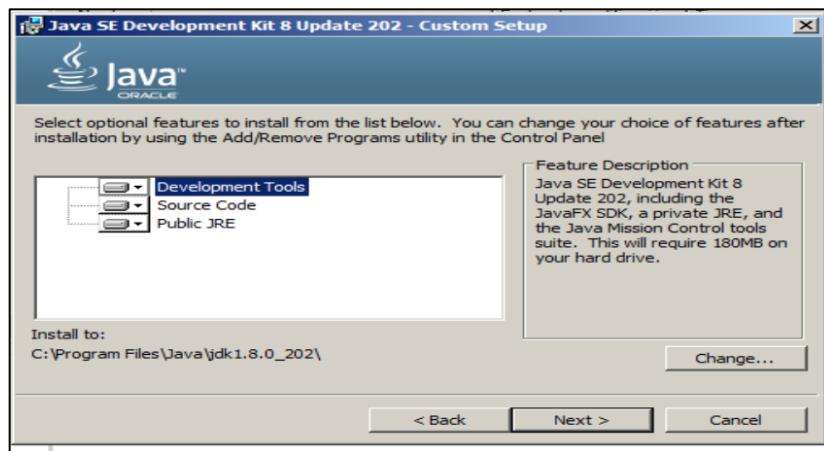


Ilustración 9. Instalación JDK 2

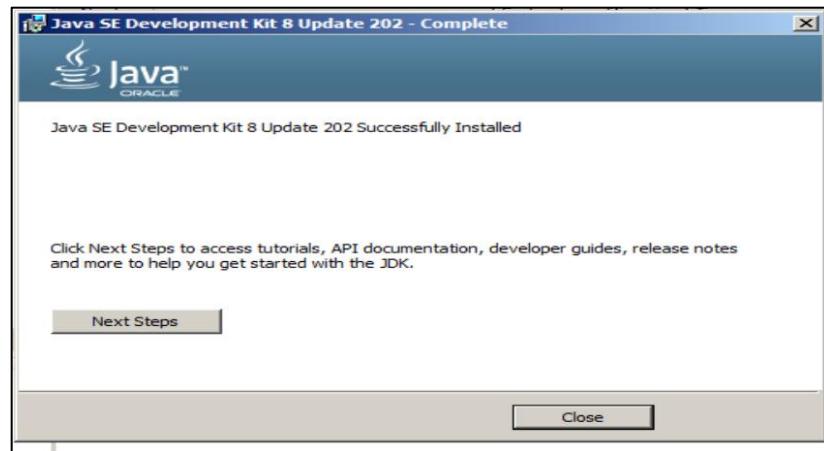


Ilustración 10. Fin instalación JDK

Una vez instalado debemos de modificar dos variables de entorno para que Geany encuentre la MVJ y el compilador, así como la carpeta de clases.

1. Copiamos la ruta hasta la carpeta bin del JDK:

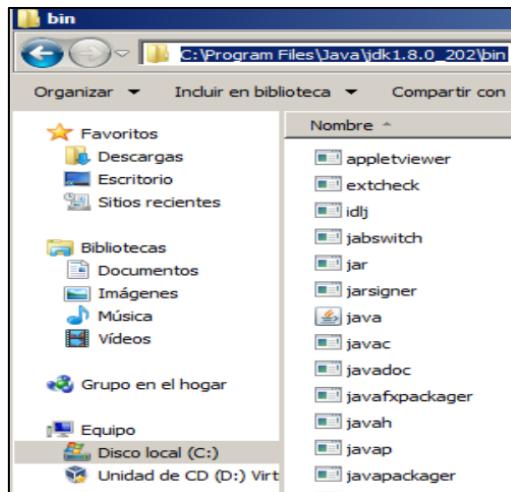


Ilustración 11. Carpeta BIN

2. Entramos en las variables de entorno de Windows y la añadimos al path:

Para acceder pulsamos el botón derecho sobre propiedades del equipo.

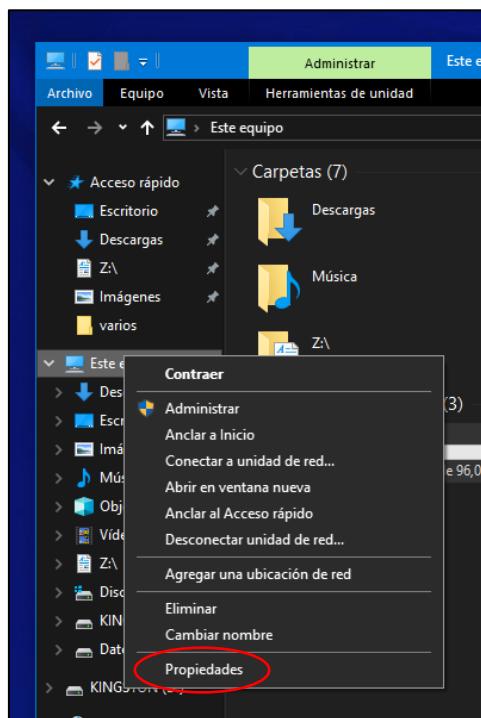


Ilustración 12. Propiedades equipo

Seleccionamos la configuración avanzada del sistema:

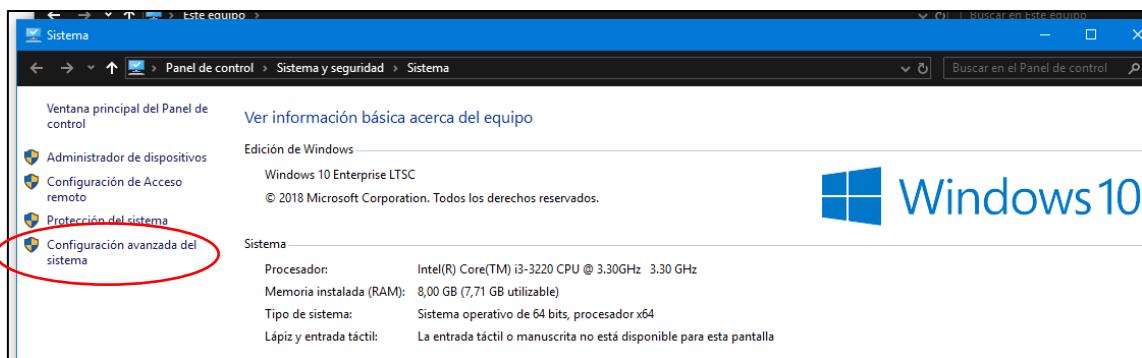


Ilustración 13. Configuración avanzada

Pulsamos en variables de entorno:

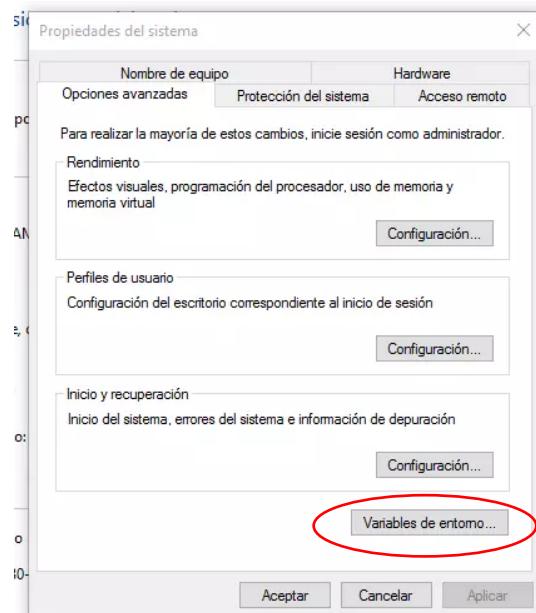


Ilustración 14. Variables de entorno

Seleccionamos la variable path y la editamos:

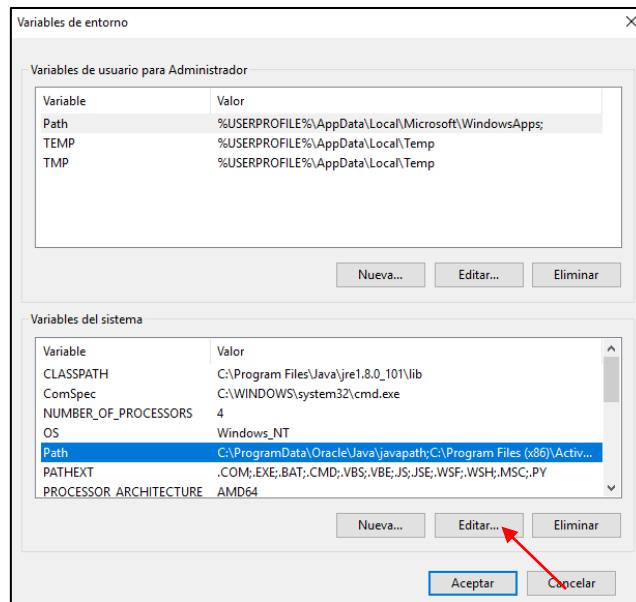


Ilustración 15. Variables de entorno 2

Creamos una nueva y pegamos la ruta que habíamos copiado

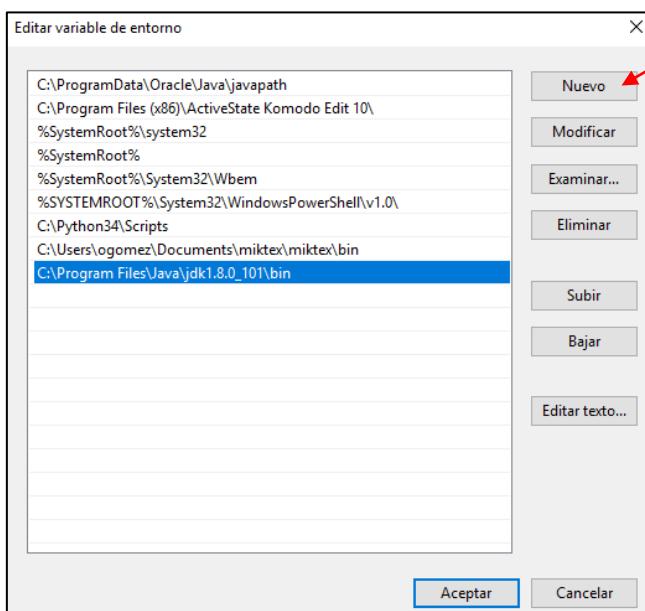


Ilustración 16. Añadir variable de entorno

Ahora solo nos queda indicar la carpeta de librerías. Si la variable CLASSPATH no existe la creamos (lo normal) si existe añadimos la carpeta lib. En este caso con poner un simple ‘.’ es suficiente, Geany la encontrará.

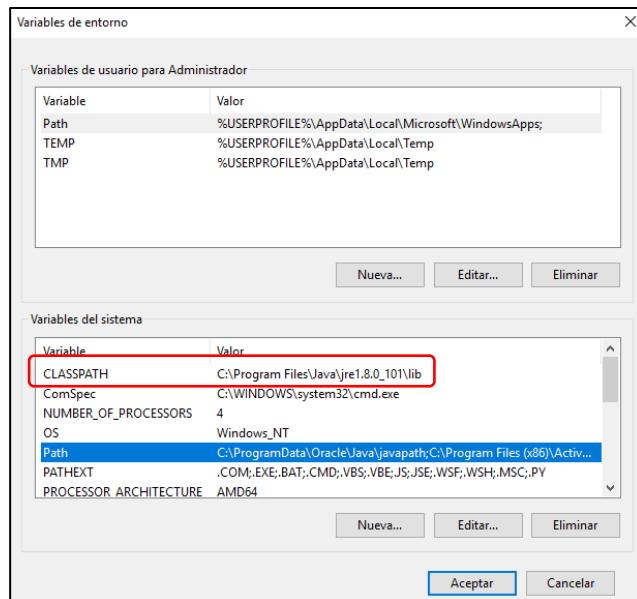


Ilustración 17. Classpath

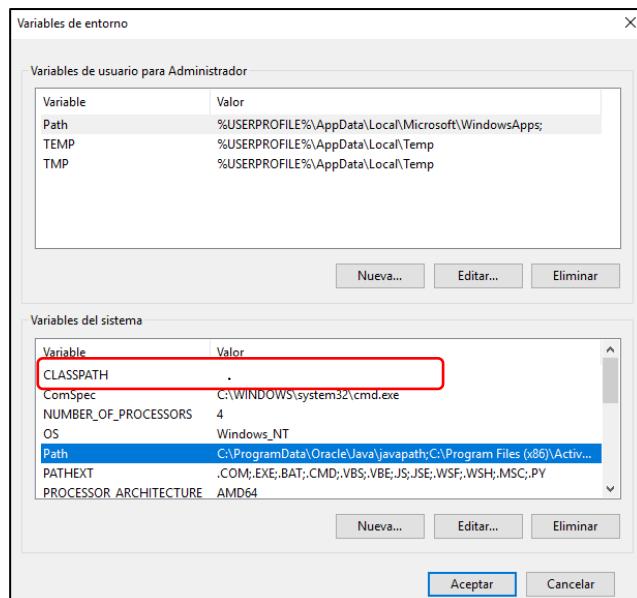


Ilustración 18. Classpath 2

### 5.1.2 Geany

Para instalar el IDE que usaremos lo primero es descargar la última versión:

<https://www.geany.org/download/releases/>

Aunque voy a mostrar la instalación de la versión 1.34 a fecha de la publicación de este manual ya se encuentra disponible la 1.38, eso sí, solo con versión de 64 bits.

La ejecutamos:



Ilustración 19. Instalación de Geany

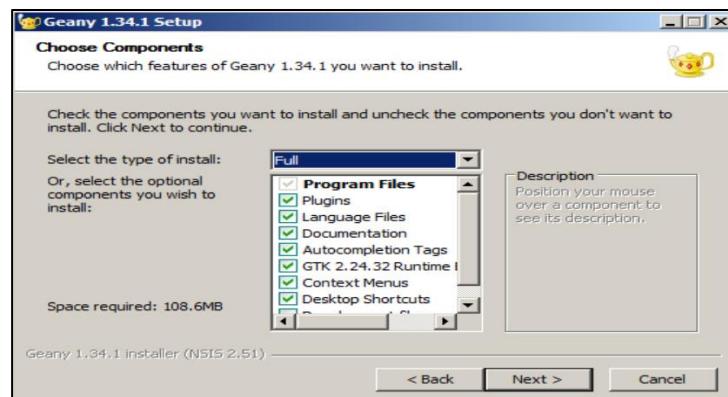


Ilustración 20. Instalación de Geany 2



Ilustración 21. Icono de Geany

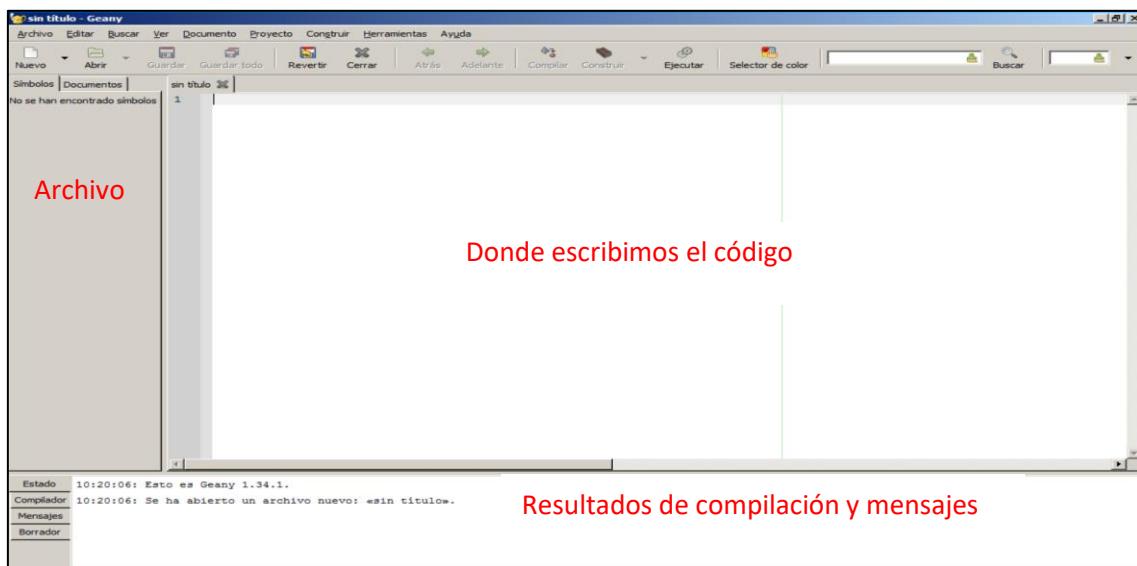


Ilustración 22. Entorno del IDE

Para comprobar que se ha instalado correctamente vamos a comprobar los comandos de ejecución de Geany. Creamos un fichero java nuevo y comprobamos en el menú correspondiente:

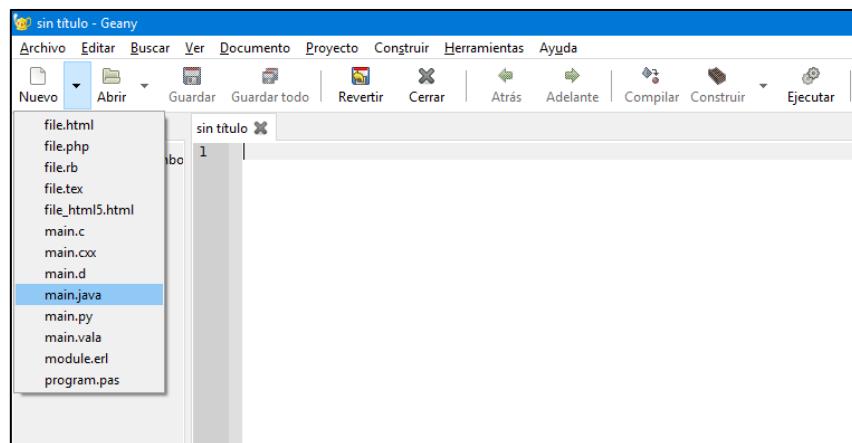


Ilustración 23. Crear una nueva clase

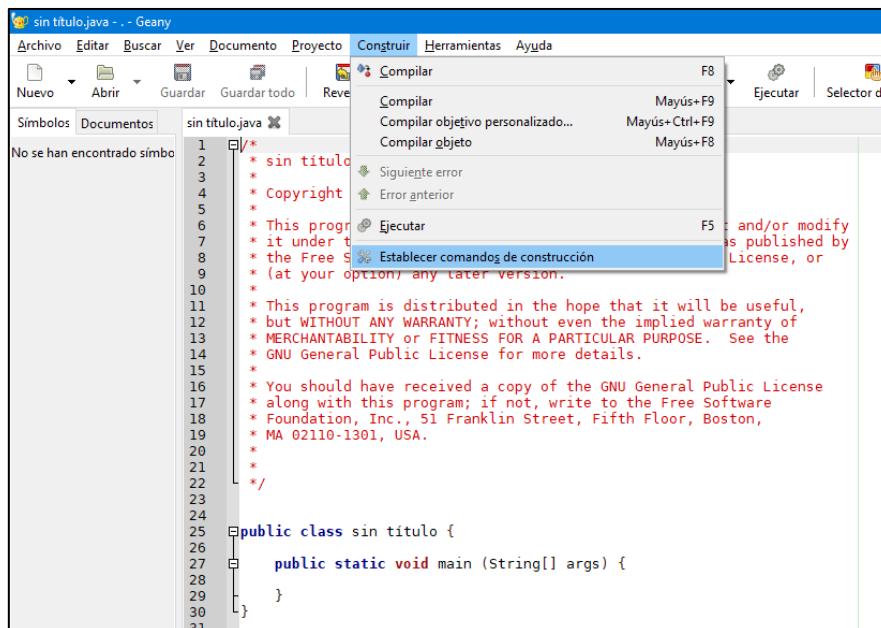


Ilustración 24. Comandos de construcción

Podemos ver como en los comandos de ejecución de llama al comando javac para compilar y java para ejecutar. Todo está correcto.

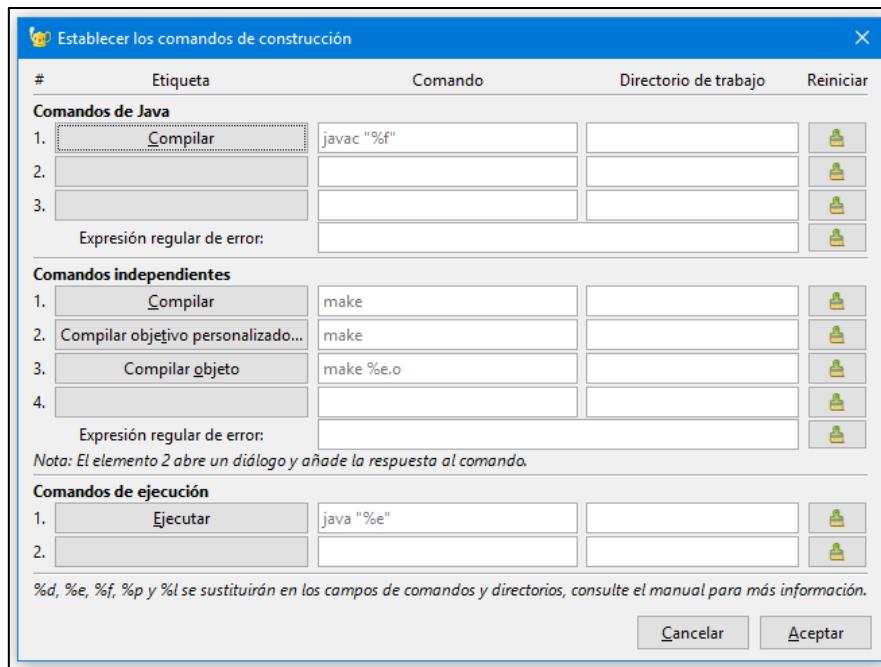


Ilustración 25. Configuración de los comandos

## 6. Comenzando

Como ya hemos visto Java es un lenguaje orientado a objetos. En este manual no voy a comenzar a explicar directamente POO sino que vamos a realizar una serie de aproximaciones hasta llegar a la POO pura. Primero usaremos Java como si fuera un lenguaje sin métodos o funciones y aprenderemos las estructuras básicas de la programación estructurada “cuadriculando” nuestra mente, pensando en soluciones a problemas desde un punto de vista algorítmico. Luego introduciremos el concepto de métodos, clases y objetos acercándonos ya a la POO. Una vez visto esto añadiremos los conceptos de herencia, polimorfismo, interfaces, clases abstractas y alguna estructura dinámica de la API.

Este manual no está pensado como un texto extremadamente largo de teoría sino más bien como un manual práctico que cualquiera desde cero puede seguir.

El hecho de que Java sea un leguaje orientado a objetos puro hace imposible no usar objetos desde el principio como en Python o PHP y luego ver la POO, por eso voy a hacer una introducción bastante simple del funcionamiento, que ya en la segunda parte explicaré más detenidamente. También me gustaría aclarar que NO todo lo que voy a explicar es la verdad y voy a omitir ciertos aspectos hasta su momento. ¿Por qué? Muy simple, para aquella gente que no ha programado nunca y comienza con la POO les resultan confusos algunos términos, estos los voy explicando poco a poco formando un circulo virtual que sólo cuando se cierra se tiene una comprensión completa.

Como ya he dicho sólo hay una forma de aprender a programar: haciendo muchos ejercicios y enfrentándote a ellos sin copiar la solución. Debéis de tener en cuenta que a la hora de aprender a programar tenéis dos problemas: el primero es conseguir la idea feliz que resuelve el problema (cuadricular la mente) y el segundo es aprender la sintaxis del lenguaje que vais a usar, en este caso Java. En cada capítulo os dejaré ejercicios para practicar. ¿Cómo afrontar un problema? Con lápiz y papel. Primero debéis de haceros un esquema en papel, usando diagramas, pseudocódigo o Java de cómo pensáis resolver el ejercicio. Despues repasad la hoja, encontrareis errores, corregidlos. Solo cuando hayáis pasado estas dos fases podéis codificar el problema en el ordenador. Es imposible codificar la solución a un problema si no sabéis como abordarlo.

Todos mis exámenes, de programación I, son escritos, en papel. Un programador debe de saber lo que puede o no funcionar sin probarlo, por lo menos lo más importante, si realizamos todos los ejercicios con el ordenador pueden funcionar por prueba y error. Debemos de ser programadores no codificadores de códigos que vamos encontrando por internet formando algo totalmente ilegible.

Bien, como suele ser habitual comencemos por hacer el famoso “hola mundo” y a partir de ahí iré explicando las cosas que debemos de saber desde el principio.

La forma más rápida para crear un programa en java es partir de la pestaña *nuevo* seleccionando main.java:

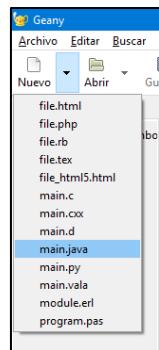


Ilustración 26. Creación de un nuevo programa

```

1 /**
2 * Holamundo.java
3 *
4 * Copyright 2022 Usuario <Usuario@DESKTOP-A0UEQ52>
5 *
6 * This program is free software; you can redistribute it and/or modify
7 * it under the terms of the GNU General Public License as published by
8 * the Free Software Foundation; either version 2 of the License, or
9 * (at your option) any later version.
10 *
11 * This program is distributed in the hope that it will be useful,
12 * but WITHOUT ANY WARRANTY; without even the implied warranty of
13 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
14 * GNU General Public License for more details.
15 *
16 * You should have received a copy of the GNU General Public License
17 * along with this program; if not, write to the Free Software
18 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
19 * MA 02110-1301, USA.
20 *
21 */
22
23 public class Holamundo {
24
25     public static void main (String[] args) {
26
27         System.out.println("Hola Mundo");
28
29     }
30 }
31
32
33

```

Estado 12:41:12: Esto es Geany 1.35.  
 Compilador 12:41:12: Se ha abierto un archivo nuevo: «sin título».  
 Mensajes 12:41:14: El archivo «sin título» ha sido cerrado.  
 Borrador 12:43:33: Se ha abierto un archivo nuevo: «sin título.java».  
 12:46:14: Archivo C:\Users\Usuario\Desktop\Holamundo.java guardado.

Ilustración 27. Código predefinido

En el panel de la izquierda podemos ver las clases que tenemos abiertas junto con la línea en la que se declara y donde se encuentra el *main*.

En java todo el código se crea en unas “plantillas” llamadas *clases*<sup>1</sup>. Estas clases se tienen que corresponder con un fichero. El fichero y la clase se deben de llamar igual. El *main* es un método especial de Java, de momento le llamaremos “programa principal”. Todo lo que queramos que Java ejecute debe de estar dentro del *main*.

En la línea 24 se declara la clase usando la siguiente sintaxis: la palabra reservada<sup>2</sup> *public*, indicando que todo el mundo lo puede ejecutar, seguida de *class* que indica que estamos definiendo una clase cuyo nombre aparece justo a su derecha.

1. En el panel horizontal de abajo sale lo que ha sucedido: Se ha abierto un fichero, se ha guardado con el nombre *Holamundo*...
2. En la línea 26 se declara el *main* o programa principal siempre comienza por *public* (todo el mundo lo puede ejecutar) seguido de *static final*, que explicaré en próximos capítulos (de momento se pone siempre). Después aparece en rojo la palabra *void* (sin retorno, lo veremos más adelante) seguida de *main*. Siempre debe de llamarse *main* si no Java no sabrá que ejecutar. Después del *main* vienen unos argumentos entre paréntesis que entenderéis más adelante. Lo que se encuentra entre llaves en el *main* es lo que Java ejecuta.
3. En la línea 28 se encuentra el código que muestra por pantalla la frase “hola mundo”. Ese código lo veremos más adelante, pero veamos lo que sucede.



Ilustración 28. Botones de Geany

Primero pulsamos en el botón compilar para que Java compruebe si hay errores de sintaxis y si todo está correcto.

```
Estado   javac "Holamundo.java" (en el directorio: C:\Users\Usuario\Desktop)
Compilador La compilación ha terminado con éxito.
Mensajes
Borrador
```

Ilustración 29. Compilación

En la pestaña compilador nos aparece la orden *javac* y nos dice que ha compilado correctamente sin encontrar ningún problema. Si pulsamos ahora en ejecutar lo ejecutará

<sup>1</sup> Veremos esto con más detenimiento en la segunda parte del manual

<sup>2</sup> Una palabra reservada es una palabra que tiene sentido para Java y que el compilador entiende. En Geany suelen pintarse de azul.

mediante el comando `java` mostrando el resultado en una ventana de consola y esperando hasta pulsar intro o que la cerremos.

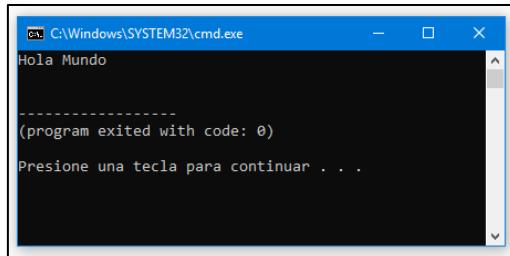


Ilustración 30. Resultado de ejecución

Se muestra el resultado de la ejecución que no es más que mostrar por pantalla la frase "Hola Mundo".

#### Expliquemos un poco más el código:

Lo primero que nos llama la atención son un montón de líneas rojas. Esas líneas son comentarios que añade Geany al crear un fichero nuevo usando el método que os he comentado, las podemos borrar. Pero, ¿Qué es un comentario? Un comentario es cualquier código al cual el compilador no hará ni caso. Sirve para poner anotaciones que nos ayuden a entender el código más adelante o a que lo entienda otra persona. Los comentarios en Java se escriben de la siguiente forma:

- Usando `//` lo que hace que toda la línea sea un comentario
- Usando `/* */` lo que hace que lo que se encuentra entre esos caracteres se tome como un comentario.

Veamos un ejemplo:

 A screenshot of a Java code editor showing a file named 'Holamundo.java'. The code contains several red comments: line 1 has a multi-line comment starting with `/*` and ending with `*/`; line 8 has a single-line comment starting with `//`. The code defines a class 'Holamundo' with a main method that prints 'Hola Mundo' to the console.
 

```

1  /* este programa
2   * escribe por pantalla una frase*/
3
4  public class Holamundo {
5
6      public static void main (String[] args) {
7
8          //imprime hola mundo
9          System.out.println("Hola Mundo");
10         //System.out.println("saludos");
11     }
12 }
  
```

Ilustración 31. Comentarios

El compilador no va a ejecutar la línea 1, 2, 8 ni 10. Fíjate que la línea 10 pese a que tiene código válido el compilador no la ejecuta al ser un comentario.

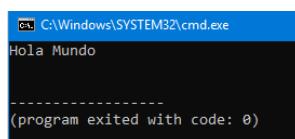


Ilustración 32. Salida comentarios

De esto podemos deducir que si estamos intentando buscar un error podemos comentar líneas para que no se ejecuten en vez de borrarlas y tenerlas que volver a escribir. Cuando en Geany usamos la forma multilínea y pulsamos intro en cada una suele añadir un \* para resaltarla.

```

1  /* este programa
2   * escribe por
3   * pantalla una frase*/
4
5  public class Holamundo {
6
7      public static void main (String[] args) {
8
9          //imprime hola mundo
10         System.out.println("Hola Mundo");
11         //System.out.println("saludos");
12     }
13 }
14
15

```

Ilustración 33. Comentarios 2



Hay autores/profesores que dicen que el uso de comentarios está mal porque el código debe de ser lo suficientemente comprensible sin la necesidad de apuntes externos ni comentarios. Yo no voy a entrar en discusión, cada cual que haga lo que quiera, pero sí os voy a decir que el uso de comentarios os sacará de más de un apuro cuando después de tres años de haber hecho un módulo tengáis que cambiar algo o cuando tengáis que modificar código que no es vuestro. Obviamente todo tiene su punto, no me parece bien el no usarlos pero tampoco llenar el código de comentarios inútiles. “aquí declaro una variable entera” a ver, hasta ahí llega cualquier programador.

Otro símbolo que podemos ver en el código son las llaves {}. Todo lo que se encuentra entre llaves es un bloque de código. De momento nos sobra con saber que después de declarar una clase se abren llaves al igual que con el *main*, al finalizar se cierran.

```

3  public class Holamundo {
4
5      public static void main (String[] args) {
6
7          System.out.println("Hola Mundo");
8
9      }
10 }
11

```

Ilustración 34. Bloque de código

La llave de la línea 9 cierra la de la 5 y la de la 10 cierra la de la clase, 3.

En Java todas las líneas, excepto declaraciones y otras especiales que veremos, terminan con ;

```
3  public class Holamundo {  
4      public static void main (String[] args) {  
5          System.out.println("Hola Mundo");  
6          System.out.println("Hola Mundo");  
7          System.out.println("Hola Mundo");  
8      }  
9  }
```

Ilustración 35. Punto y coma



Pese que no forma parte de la sintaxis de Java es de buena praxis y denota conocimiento del lenguaje que las clases comiencen por mayúsculas.



Java es sensible a las mayúsculas, por tanto no es lo mismo A que a, tanto en el código como en el nombre de los ficheros y clases, diga lo que diga Windows.

## 7. Conceptos iniciales

Una de las primeras cosas que podemos realizar en un programa son operaciones. Veamos un ejemplo de una suma:

```

3  public class Operaciones {
4
5     public static void main (String[] args) {
6
7         System.out.println(2+3);
8
9     }
10}

```

Ilustración 36. Primer ejemplo

Estamos mostrando por pantalla el resultado de sumar 2 y 3, se mostrará 5.

Los números aparecen en verde porque son literales: el número 2 y el número 3, y siempre serán esos dos números. En Java los literales pueden ser tipos de datos primitivos, cadenas de texto o el valor null<sup>3</sup>.

Los tipos primitivos son tipos de datos que forman parte del propio lenguaje, están definidos en él y no necesitan de importar ninguna librería. Los tipos primitivos de Java son:

Dato	Tipo	Bits	Rango
carácter	char	16	0 a 65535
entero	byte	8	-128 a 127
	short	16	-32768 a 32767
	int	32	-2147483648 a 2147483647
	long	64	-9223372036854775808 a 9223372036854775807
real	float	32	-3.4x10 <sup>38</sup> a -1.4x10 <sup>-45</sup> , 1.4x10 <sup>-45</sup> a 3.4x10 <sup>38</sup>
	double	64	-1.7x10 <sup>308</sup> a -4.9x10 <sup>-324</sup> , 4.9x10 <sup>-324</sup> a 1.7x10 <sup>308</sup>
booleano	boolean	1	true, false

Ilustración 37. Tipos de datos

Tenemos cuatro tipos de enteros y dos tipos de números reales o decimales. Cuanto más bits más grandes pueden ser.

Como podemos ver un carácter no es más que un entero de tipo short, de ahí que sea muy fácil pasar de carácter a entero y viceversa. Un carácter no es más que un símbolo asociado a un número entero. En Java se usa Unicode no obstante voy a mostrar el ejemplo con la tabla ASCII ya que es más fácil de entender y es totalmente compatible:

---

<sup>3</sup> Lo veremos más adelante

The ASCII code														
ASCII control characters			ASCII printable characters						Extended ASCII characters					
DEC	HEX	Símbolo ASCII	DEC	HEX	Símbolo	DEC	HEX	Símbolo	DEC	HEX	Símbolo	DEC	HEX	Símbolo
00 00h	NULL	(carácter nulo)	64 40h	@	96 60h	'	128 80h	Ç	160 A0h	â	192 C0h	l	224 E0h	Ó
01 01h	SOH	(inicio encabezado)	65 41h	A	97 61h	a	129 81h	é	162 A2h	í	193 C1h	ł	225 E1h	à
02 02h	STX	(inicio texto)	66 42h	B	98 62h	b	130 82h	ê	162 A2h	ó	194 C2h	ł	226 E2h	Ó
03 03h	ETX	(fin de texto)	67 43h	C	99 63h	c	131 83h	á	163 A3h	ú	195 C3h	ł	227 E3h	Ó
04 04h	EOT	(fin transmisión)	68 44h	D	100 64h	d	132 84h	á	164 A4h	ñ	196 C4h	ł	228 E4h	ó
05 05h	ENQ	(enquiry)	69 45h	E	101 65h	e	133 85h	à	165 A5h	ñ	197 C5h	+	229 E5h	Ó
06 06h	ACK	(acknowledgement)	70 46h	F	102 66h	f	134 86h	â	166 A6h	»	198 C6h	à	230 E6h	µ
07 07h	BEL	(timbre)	71 47h	G	103 67h	g	135 87h	ç	167 A7h	º	199 C7h	À	231 E7h	þ
08 08h	BS	(retroceso)	72 48h	H	104 68h	h	136 88h	é	168 A8h	ç	200 C8h	æ	232 E8h	þ
09 09h	HT	(tab horizontal)	73 49h	I	105 69h	i	137 89h	è	169 A9h	ø	201 C9h	ł	233 E9h	ú
10 0Ah	LF	(salto de linea)	74 4Ah	J	106 6Ah	j	138 8Ah	ê	170 AAh	¬	202 CAh	ł	234 EAh	Ü
11 0Bh	VT	(tab vertical)	75 4Bh	K	107 6Bh	k	139 8Bh	í	171 A Bh	½	203 C Bh	ł	235 EBh	Ü
12 0Ch	FF	(form feed)	76 4Ch	L	108 6Ch	l	140 8Ch	í	172 ACh	¼	204 CCh	ł	236 EC h	ý
13 0Dh	CR	(retorno de carro)	77 4Dh	M	109 6Dh	m	141 8Dh	í	173 ADh	í	205 CDh	ł	237 EDh	Y
14 0Eh	SO	(shift Out)	78 4Eh	N	110 6Eh	n	142 8Eh	â	174 AEh	«	206 CEh	ł	238 EEh	-
15 0Fh	SI	(shift In)	79 4Fh	O	111 6Fh	o	143 8Fh	À	175 AFh	»	207 CFh	ł	239 EFh	-
16 10h	DLE	(data link escape)	80 50h	P	112 70h	p	144 90h	É	176 B0h	ø	208 D0h	ð	240 F0h	-
17 11h	DC1	(device control 1)	81 51h	Q	113 71h	q	145 91h	æ	177 B1h	ł	209 D1h	ð	241 F1h	±
18 12h	DC2	(device control2)	82 52h	R	114 72h	r	146 92h	Æ	178 B2h	ł	210 D2h	ð	242 F2h	-
19 13h	DC3	(device control3)	83 53h	S	115 73h	s	147 93h	ó	179 B3h	ł	211 D3h	ð	243 F3h	½
20 14h	DC4	(device control4)	84 54h	T	116 74h	t	148 94h	ò	180 B4h	ł	212 D4h	ð	244 F4h	¾
21 15h	NAK	(negative acknowle-	85 55h	U	117 75h	u	149 95h	ó	181 B5h	À	213 D5h	ł	245 F5h	§
22 16h	SYN	(synchronous idle)	86 56h	V	118 76h	v	150 96h	ú	182 B6h	Ã	214 D6h	ł	246 F6h	÷
23 17h	ETB	(end of trans. block)	87 57h	W	119 77h	w	151 97h	û	183 B7h	ä	215 D7h	ł	247 F7h	-
24 18h	CAN	(cancel)	88 58h	X	120 78h	x	152 98h	ÿ	184 B8h	ø	216 D8h	ł	248 F8h	¤
25 19h	EM	(end of medium)	89 59h	Y	121 79h	y	153 99h	ø	185 B9h	ł	217 D9h	ł	249 F9h	-
26 1Ah	SUB	(substitute)	90 5Ah	Z	122 7Ah	z	154 9Ah	Ù	186 BAh	ł	218 DAh	ł	250 FAh	-
27 1Bh	ESC	(escape)	91 5Bh	[	123 7Bh	{	155 9Bh	ø	187 BBh	ł	219 DBh	ł	251 FBh	ı
28 1Ch	FS	(file separator)	92 5Ch	\	124 7Ch		156 9Ch	£	188 BC h	ł	220 DC h	ł	252 FC h	¤
29 1Dh	GS	(group separator)	93 5Dh	]	125 7Dh	}	157 9Dh	Ø	189 BDh	¢	221 DDh	ł	253 FDh	¤
30 1Eh	RS	(record separator)	94 5Eh	^	126 7Eh	~	158 9Eh	×	190 BEh	¥	222 DEh	ł	254 FEh	■
31 1Fh	US	(unit separator)	95 5Fh	-	127 7Fh	-	159 9Fh	f	191 BFh	ł	223 DFh	ł	255 FFh	-
127 20h	DEL	(delete)					theASCIIcode.com.ar							

Ilustración 38. Tabla ASCII

Como podéis ver al carácter 'A' le corresponde el número 65, a la 'a' el 97 y luego hay una serie de caracteres "extraños" y no imprimibles que también tienen su equivalencia en entero.

**En Java un carácter siempre va entre comillas simples: 'a'**

Existe un tipo especial y muy importante en programación llamado booleano y que solo puede contener dos posibles valores: verdadero o falso. No puede haber otra opción. En Java se escriben en inglés: true o false.

En Java existe un tipo de datos NO primitivo, realmente es una clase pero se usa tanto que quiero introducirlo aquí y es el *String* (la S en mayúscula). Esta clase es tan usada que Java permite crear *String's* de una forma similar a un tipo primitivo. Un *String* no es más que una cadena de texto (uno o más caracteres) y se ponen entre **comillas dobles**, por ejemplo “**hola mundo**”.

**Caracteres de escape:** Java al igual que otros lenguajes tiene una serie de caracteres especiales que se pueden usar anteponiendo el carácter de escape \ (contrabarra):

Secuencia de escape	Descripción
\t	Un tabulador
\b	Un retroceso
\n	Un carácter de nueva línea
\r	Un retorno de carro
\f	Un salto de línea
\'	Una comilla simple
\\"	Una comilla doble
\\\	Un carácter de barra invertida

Tabla 1. Caracteres de escape

Veamos un ejemplo con el más usado que es nueva línea y de paso comprenderemos mejor la línea que uso para imprimir por pantalla.

Pensemos que en Java existe “algo” llamado *System* que nos permite hacer algunas cosas. Una de ellas es la comunicación con el exterior existiendo tres subapartados de ese *System*:

- **in**: Entrada estándar. De forma predefinida el teclado.
- **out**: Salida estándar. De forma predefinida la pantalla.
- **err**: Salida estándar para errores. De forma predefinida la pantalla.

Por tanto, cuando usamos *System.out* lo que estamos diciendo es que dirija la información a la pantalla. Dentro de ese *System.out* existen una serie de métodos<sup>4</sup> que nos permiten imprimir por pantalla, uno de ellos el *println*. Veamos:

```

2  public class Operaciones {
3
4      public static void main (String[] args) {
5
6          //escribe el texto y luego inserta una linea nueva (\n)
7          System.out.println("hola");
8          //escribe el texto y luego no hace nada mas
9          System.out.print("Pepe");
10         System.out.print("Pedro");
11         //escribe el texto junto al caracter de escape
12
13         System.out.print("Eva\n");
14         System.out.print("Pablo\n");
15
16     }
17 }
```

Ilustración 39. Nueva línea

```

C:\Windows\SYSTEM32\cmd.exe
hola
PepePedroEva
Pablo

```

Ilustración 40. Ejemplo nueva línea

- Línea 7: Muestra “hola” y luego introduce una nueva línea al usar *println*.
- Línea 9: Muestra “Pepe” y luego no introduce una nueva línea al usar *print*.
- Línea 10: Muestra “Pedro” y luego no introduce una nueva línea al usar *print*. Como antes no hemos introducido una línea nueva se escribe en la misma sin separación.
- Línea 13: Muestra “Eva” y luego introduce una nueva línea al usar “\n”. Como antes no hemos introducido una línea nueva se escribe en la misma sin separación.
- Línea 14: Muestra “Pablo” y luego introduce una nueva línea al usar “\n”. Como antes hemos introducido una línea nueva se escribe debajo.

Cuando Java ve la contrabarra sabe que viene un carácter especial y lo interpreta. Los caracteres especiales como son texto también se pueden concatenar como luego veremos.

---

<sup>4</sup> Un programita que ya viene hecho en Java y que podemos usar.

**Cuidado:**

```

3  public class Operaciones {
4
5      public static void main (String[] args) {
6          byte x=55;
7
8          System.out.println(x);
9
10 }

```

*Ilustración 41. Ejemplo rango de datos*

Esto funcionará sin ningún problema, pero lo siguiente no compilará pues un byte no tiene suficiente precisión como para guardar el número 300:

```

3  public class Operaciones {
4
5      public static void main (String[] args) {
6          byte x=300;
7
8          System.out.println(x);
9
10 }

```

*Ilustración 42. Ejemplo rango de datos 2*

Como hemos visto en la tabla de tipos el byte sólo llega hasta 127.

### 7.1 Variables

Antes hemos hablado de literales, pero obviamente no tiene sentido hacer un programa que sólo sepa sumar 2+3, para ello existen las variables. Las variables no son más que porciones de memoria a las que Java da un nombre y que guardan un dato.

Para declarar una variable primero se pone el tipo al que pertenece y luego su nombre.

*<tipo> nombre;*

Reglas a tener en cuenta antes de elegir el nombre de una variable:

- Una variable, siempre debe iniciar con una letra (mayúscula o minúscula) o un guión bajo (\_).
- Una variable puede contener números siempre que no comience por uno.
- El nombre de una variable no puede contener espacios en blanco.
- No puedes utilizar palabras reservadas<sup>5</sup> para la declaración de una variable.
- Una variable no puede cambiar de nombre.
- No pongáis acentos.

**Regla de oro:** No te compliques la vida.

---

<sup>5</sup> Palabras que tienen un significado para Java



- Las variables deben de tener nombres representativos.
- En Java las variables comienzan en minúsculas y si son nombres compuestos no se usa el \_ para separarlos, cada comienzo de la nueva palabra se pone en mayúsculas.
- En Java las variables primitivas no se inicializan automáticamente a ningún valor.

#### Ejemplo de nombres validos:

- nombrePaterno
- IdCliente
- numero8
- \_conteoCiclos

#### Ejemplo de nombres no validos:

- 3puesto
- numero Telefono
- int

#### Aclaremos esto con ejemplos:

##### 1. Error de compilación ya que las variables no han sido inicializadas

```

3  public class Operaciones {
4
5      public static void main (String[] args) {
6          int x;
7          int y;
8          int suma;
9          suma=x+y;
10
11         System.out.println(suma);
12
13     }
14 }
```

Ilustración 43. Variables no inicializadas

<pre> Estado  javac "Operaciones.java" (en el directorio: C:\Users\Usuario\Desktop) compilador Operaciones.java:9: error: variable x might not have been initialized                                 suma=x+y;  ^ Operaciones.java:9: error: variable y might not have been initialized                                 suma=x+y;  ^ 2 errors Ha fallado la compilación. </pre>
---

Ilustración 44. Variables no inicializadas 2

Dos posibles soluciones: darles valor después de declararlas o en la propia declaración:

```

3  public class Operaciones {
4
5    public static void main (String[] args) {
6      int x;
7      int y;
8      x=2;
9      y=3;
10     int suma;
11     suma=x+y;
12
13     System.out.println(suma);
14
15   }
16

```

Ilustración 46. Variables inicializadas

```

3  public class Operaciones {
4
5    public static void main (String[] args) {
6      int x=2;
7      int y=3;
8      int suma;
9      suma=x+y;
10
11     System.out.println(suma);
12
13   }
14

```

Ilustración 45. Variables inicializadas 2

El símbolo = se llama asignación y lo que hace es darle el valor que se encuentra a la derecha del mismo a la variable que se encuentra a la izquierda.

Si hacemos una traza mirando lo que ocurre en memoria de la solución de la derecha entenderemos como funciona:

Dirección de memoria	Nombre de la variable	Contenido	Línea de código
0x00F2	x		6
0x00F4	y		7
0x00F6			
0x00F8			

Tabla 1. Variables, memoria 1

Dirección de memoria	Nombre de la variable	Contenido	Línea de código
0x00F2	x	2	6
0x00F4	y	3	7
0x00F6			
0x00F8			

Tabla 2. Variables, memoria 2

Dirección de memoria	Nombre de la variable	Contenido	Línea de código
0x00F2	x	2	6
0x00F4	y	3	7
0x00F6	suma		10
0x00F8			

Tabla 3. Variables, memoria 3

Dirección de memoria	Nombre de la variable	Contenido	Línea de código
0x00F2	x	2	6
0x00F4	y	3	7
0x00F6	suma	5	10
0x00F8			

Tabla 4. Variables, memoria 4

Varias variables también se pueden declarar en una misma línea: int x,y,z; o int x=2, y=3;

1. Declaración de los distintos tipos de variable:

```

3  public class Operaciones {
4
5   public static void main (String[] args) {
6     int numero=2;
7     char letra='a';
8     double decimal=4.5;
9     boolean logico=true;
10    String texto="hola mundo";
11
12    System.out.println(numero);
13    System.out.println(letra);
14    System.out.println(decimal);
15    System.out.println(logico);
16    System.out.println(texto);
17
18  }
19

```

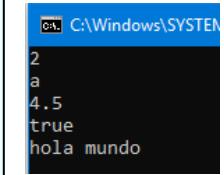


Ilustración 47. Tipos de variable 2

Ilustración 48. Tipos de variable



En Java no hay '' decimal sino '' decimal  
El String no aparece en rojo ya que no es un tipo primitivo.

2. Ejemplos de tipos de variable:

char letra='aa';	Esto dará un error de compilación ya que un char sólo puede contener un carácter.
String letra="a";	Aunque solo contenga una letra sigue siendo un String, una cadena de texto.
String letra="5";	Esto es el símbolo 5 no el número 5, por tanto "5"+2 no es 7

Tabla 5. Ejemplos de tipos letra



Las variables solo son visibles dentro del bloque en el que se han declarado, una vez termina este mueren. Un bloque es el código encerrado entre llaves.



En Java existen múltiples métodos para convertir un tipo en otro. Los veremos más adelante, pero uno de los más usados y que ya debéis de conocer es Integer.parseInt(). Integer es una clase de la API y parseInt() un método de la misma. Dado un String devuelve un entero. Por ejemplo:

```
int x=Integer.parseInt("5");
```

Transforma el texto 5 en el número 5 y lo guarda en x. El primero no se puede ni sumar ni multiplicar, el segundo sí ya que es un número.

## 7.2 Constantes

Java no tiene constantes como tal, pero se pueden simular usando unos modificadores que veremos más adelante, pero de momento debéis de saber que podéis declarar una variable cuyo contenido una vez asignado no pueda ser cambiado nunca, por tanto, se vuelve constante. La sintaxis es:

*final <tipo> nombre=valor;*

```
3  public class Operaciones {
4
5    public static void main (String[] args) {
6      final double PI=3.1416;
7
8      System.out.println(PI);
9
10     }
11 }
```

Ilustración 49. Constantes

Si después de la declaración de intenta modificar PI dará un error de compilación.

**Las constantes en Java se escriben en mayúsculas**



Suele ser habitual declarar las variables y constantes al principio del programa para una mejor claridad. No obstante no es obligatorio.

### 7.3 Operaciones

Entre las operaciones aritméticas que podemos realizar con datos primitivos están:

+	Operador de Suma
-	Operador de Resta
*	Operador de Multiplicación
/	Operador de División
%	Operador de Resto

Tabla 6. Operadores aritméticos

```

3 public class Operaciones {
4
5     public static void main (String[] args) {
6         int numero1=7;
7         int numero2=5;
8
9         System.out.println(numero1+numero2);
10        System.out.println(numero1-numero2);
11        System.out.println(numero1*numero2);
12        System.out.println(numero1/numero2);
13        System.out.println(numero1%numero2);
14    }
15 }
```

Ilustración 50. Operadores aritméticos

```

C:\Windows\SYSTEM32\cmd.exe
12
2
35
1
2
```

Ilustración 51. Salida operadores

Veamos cómo se han ejecutado:

- La suma la ha realizado bien  $7+5=12$ .
- La resta también:  $7-5=2$ .
- La multiplicación no hay problema:  $7\times5=35$  (fijaros que el signo es \*).
- La división:  $7/5=1$  ¿?  $7/5=1,4$  no uno. ¿Qué ha pasado? Muy simple, la división en Java es entera si ambos operandos son enteros por tanto se queda solo con la parte entera del resultado: 1.
- El % ¿Qué es eso? El % no tiene nada que ver con porcentajes, ese símbolo representa el módulo o lo que es lo mismo, el resto de una división.

$$\begin{array}{r}
 & 5 \\
 - 7 & \boxed{ } \\
 & 5 \quad 1 \\
 \hline
 & 2
 \end{array}$$

Resto

Ahora mismo no le encontrareis sentido a lo del módulo. Tiempo al tiempo, posiblemente es uno de los operadores más usados en programación.

Pero ¿Y la división? ¿Cómo lo solucionamos? La solución pasa por que uno de los operandos sea decimal, solo con que uno sea decimal Java ya actúa como división decimal. Una posible solución, de ir por casa, podría ser esta: (luego veremos una mejor)

```

3  public class Operaciones {
4
5      public static void main (String[] args) {
6          int numero1=7;
7          double numero2=5;
8
9          System.out.println(numero1+numero2);
10         System.out.println(numero1-numero2);
11         System.out.println(numero1*numero2);
12         System.out.println(numero1/numero2);
13         System.out.println(numero1%numero2);
14     }
15 }
```

Ilustración 52. División decimal

Fijaros como lo único que he hecho es cambiar uno de los números de *int* a *double*.

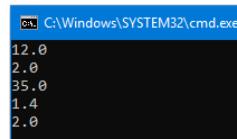


Ilustración 53. Resultado división decimal

Ya funciona correctamente pero claro ahora todos los demás cálculos añaden el 0 decimal. Luego lo solucionamos.

Hay otro operador que se suele usar mucho y es el + ¿el +? A ver José, ¿Qué te has fumado? Me has dicho que el + es suma. Sí, pero es un operador sobrecargado lo que significa que puede hacer distintas operaciones dependiendo de los tipos con los que se encuentre y si esos tipos son cadenas de texto el + significar concatenar o lo que es lo mismo unir dos textos:

```

String texto1="hola ";
String texto2="Mundo";
texto1+texto2 es igual a "Hola Mundo"
```

Veamos cómo actúa este operador con varios ejemplos:

```

3  public class Operaciones {
4
5      public static void main (String[] args) {
6          int numero1=5;
7          int numero2=2;
8          String textoNum="3";
9          String texto1="Hola";
10         String texto2="Mundo";
11
12         System.out.println(texto1+texto2);
13         System.out.println(texto1+" "+texto2);
14         System.out.println(numero1+numero2);
15         System.out.println(textoNum+numero2);
16     }
17 }
```

Ilustración 54. Concatenación

```
C:\Windows\SYSTEM32\cmd.exe
HolaMundo
Hola Mundo
7
32
```

Ilustración 55. Ejemplo concatenado

- En la línea 12 está concatenando dos *String*, los une. Pero claro el operador + solo une con lo que queda un poco feo todo junto.
- En la línea 13 lo que hacemos es concatenar un espacio entre texto1 y texto2 lo que hace que la frase ya quede mejor.
- El operador + se encuentra en la línea 14 con dos enteros por tanto actúa como suma.
- En cambio, en la línea 15 se encuentra a un lado un texto y al otro un entero, dado que en el método *println* tiene más prioridad el texto transforma automáticamente el entero en texto y los concatena.

## 7.4 Operadores de asignación

A parte de le operador de asignación = existen otros operadores que no son más que una forma recortada de los anteriores cuando se utilizan en asignaciones. Estos son:

<code>+ =</code>	Para sumar el operando izquierdo con el operando derecho y luego asignarlo a la variable de la izquierda.
<code>- =</code>	Para restar el operando izquierdo con el operando derecho y luego asignarlo a la variable de la izquierda.
<code>* =</code>	Para multiplicar el operando izquierdo con el operando derecho y luego asignándolo a la variable de la izquierda.
<code>/ =</code>	Para dividir el operando izquierdo con el operando derecho y luego asignarlo a la variable de la izquierda.
<code>% =</code>	Para asignar el módulo del operando izquierdo con el operando derecho y luego asignarlo a la variable de la izquierda.

Tabla 7. Operadores de asignación

```
3 public class Operaciones {
4
5     public static void main (String[] args) {
6         int x=1;
7         int suma=0;
8         suma=suma+1;
9
10        System.out.println(suma);
11
12    }
13}
```

Ilustración 57. Suma sin asignación

```
3 public class Operaciones {
4
5     public static void main (String[] args) {
6         int x=1;
7         int suma=0;
8         suma+=1;
9
10        System.out.println(suma);
11
12    }
13}
```

Ilustración 56. Suma con asignación

Ambos códigos hacen exactamente lo mismo. Al principio, hasta que cojáis práctica podéis usar el modo de la izquierda. El de la derecha no se ejecuta más rápido.

## 7.5 Operadores unarios aritméticos

++	incremento
--	decremento

Tabla 8. Operadores de incremento

Los operadores unarios de incremento y decremento aumentan o disminuyen el valor de una variable en una unidad, pero actúan de forma distinta dependiendo del orden:

```

3 public class Operaciones {
4
5     public static void main (String[] args) {
6         int x=1;
7
8         System.out.println(x++);
9         System.out.println(x);
10    }
11 }
```

Ilustración 58. Post incremento

```

1
2

```

Ilustración 61. Resultado Post incremento

```

3 public class Operaciones {
4
5     public static void main (String[] args) {
6         int x=1;
7
8         System.out.println(++x);
9         System.out.println(x);
10    }
11 }
```

Ilustración 59. Pre incremento

```

2
2

```

Ilustración 60. Resultado Pre incremento

Analicemos los dos casos:

- Izquierda: Al hacer `x++` primero muestra la `x` y luego la incrementa. Al mostrarla de nuevo vale 2.
- Derecha: Al hacer `++x` primero incrementa y luego muestra. Al mostrarla la segunda vez el valor es el mismo.

Habrá muchos casos en los que da igual que forma uséis, por ejemplo, yo suelo usar mucho `x++`, pero debéis de ir con cuidado porque en ocasiones no será lo mismo.

El operador `--` es igual, pero restando 1.

## 7.6 Prioridad de operadores

Todos los operadores no tienen la misma prioridad. Veamos el siguiente ejemplo:

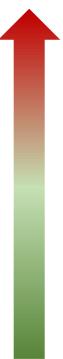
`x=5+8*2;` ¿Cuánto vale `x`? Si habéis contestado 21 es correcto. La multiplicación tiene más prioridad que la suma por lo que primero se ejecuta  $8*2=16$  y luego  $16+5=21$ .

Está claro que era un ejemplo fácil, pero ¿a qué si hubiera escrito `x=5+(8*2)` nadie hubiera dudado?

Con esto pasa como los comentarios, tan malo es pasarse como no llegar. Los paréntesis ayudan a que el código sea más legible. Está claro que si yo hubiera querido que el resultado fuera 26 tendría que haber escrito `x=(5+8)*2` ya que el operador con más prioridad es el paréntesis.

Huelga decir que si vais a hacer algo como esto uséis paréntesis: `x=4+5*6/3.7-5.9/9%67+23.`

La prioridad de la mayoría de operadores en Java es la siguiente (cuanto más rojo más prioridad)



( ) [] .
! ++ --
new (cast)
* / %
+ -
< <= > >= instanceof
== !=
&&
? :
= *= /= %= += -=

Tabla 9 Prioridad de operadores

## 7.7 Casting

El casting son conversiones que se realizan entre tipos distintos. Existen dos tipos de conversiones:

- **Conversiones implícitas:** las realiza Java de forma automática. La variable destino debe de tener más precisión que la de origen.

```

3  public class Operaciones {
4
5    public static void main (String[] args) {
6      byte x=5;
7      int y;
8      y=x;
9
10     System.out.println(y);
11   }
12 }
```

Ilustración 62. Casting implícito

Java sin decirle nada a convertido el byte a int

- **Conversiones explícitas:** Es el programador el que fuerza la conversión. Se pone el tipo destino entre paréntesis antes de la variable a convertir.

```
3  public class Operaciones {  
4  
5      public static void main (String[] args) {  
6          double x=5.6;  
7          int y;  
8          y=(int)x;  
9  
10         System.out.println(y);  
11     }  
12 }
```

Ilustración 63. Casting explícito

Hemos forzado a Java a que convierta un *double* a *int*. De esta forma podríamos solucionar el ejemplo de la división: *(double)x/y*. Con transformas uno es suficiente.



Hay que ir con mucho cuidado con los casting's. En el primero si estuviera al revés nos daría un error de compilación como vimos en el apartado de los tipos. En el segundo no dará ningún error pero la y no valdrá 5.6 sino solo 5 ya que en un int no se pueden representar decimales.

## 7.8 Ejercicios

1. Indica si funciona o no y porqué. ¿qué se mostraría?:

```
int a='a';  
  
System.out.println(a);
```

2. int pi=3.14;

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

3. Realiza un programa que dadas dos variables a y b intercambien sus valores.

4. Realiza un programa que calcule la longitud de una circunferencia de radio 3 metros.

5. Realiza un programa que calcule el área de una circunferencia de radio 5 metros.

6. Realizar un programa que dada una cantidad de dinero lo divida en billetes de 50, 20, 10, 5 y monedas de un euro (no hay céntimos)

## 8. Estructuras de control

Como hemos visto, los programas contienen instrucciones que se ejecutan una a continuación de la otra según la secuencia en la que hayamos escrito el código. Sin embargo, hay ocasiones en las que es necesario romper esta secuencia de ejecución para hacer que una serie de instrucciones se ejecuten o no dependiendo de una determinada condición o hacer que una serie de instrucciones se repitan un número determinado de veces.

Las estructuras de control permiten modificar el orden natural de ejecución de un programa. Mediante ellas podemos conseguir que el flujo de ejecución de las instrucciones sea el natural o varíe según se cumpla o no una condición o que un bloque de instrucciones se repita dependiendo de que una condición se cumpla o no.

Las estructuras de control tienen las siguientes características:

- Tienen un único punto de entrada y un único punto de salida.
- Están compuestas por instrucciones o por otras estructuras de control.

Las estructuras de control se dividen en las siguientes categorías:

- Estructura Secuencial
- Estructura Condicional o alternativa
- Estructura Repetitiva.
- Instrucciones de salto

Las primeras son las que hemos visto hasta ahora, después de la línea 1 se ejecuta la 2 y luego la 3 y así todas.

### 8.1 Estructuras condicionales

#### 8.1.1 if

El if es una estructura que nos va a permitir ejecutar un código u otro dependiendo de una condición. Si la condición es verdadera entra en el bloque de la estructura, si no, no.

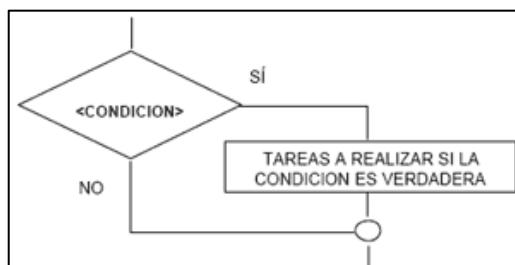


Ilustración 64.Estructura del if

Ejemplo en Java:

```

3  public class Pruebas {
4
5      public static void main (String args[]) {
6          int x=5;
7
8          if(x==5)
9              System.out.println("es un cinco");
10         System.out.println("Fin de programa");
11     }
12
13 }
14

```

Ilustración 65. Primer if

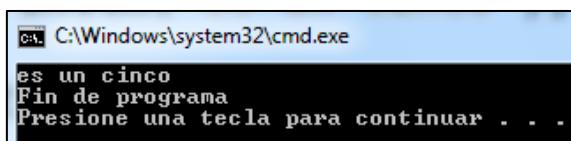


Ilustración 66. Ejecución primer if

En Java la palabra reservada para esta estructura es *if*. El *if* va seguido de la condición entre paréntesis, en este caso estamos comprobando si la *x* es igual a 5. Como es verdad se ejecuta lo que hay dentro del *if* y luego continúa con la siguiente instrucción del programa.

Imaginemos que *x* es igual a 10. El resultado sería este:

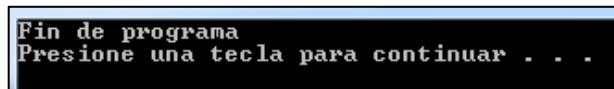


Ilustración 67. Ejecución del primer if 2

Dado que la condición es falsa no entra en el *if* y se ejecuta la siguiente sentencia del programa.

Todas las estructuras de Java solo permiten una única sentencia, para poder poner más es necesario encerrarlas en un bloque entre llaves:

```

2  public class Pruebas {
3
4      public static void main (String args[]) {
5          int x=5;
6
7          if(x==5){
8              System.out.println("la condición es verdadera");
9              System.out.println("es un cinco");
10             System.out.println("el doble de x es "+(x*2));
11         }
12         System.out.println("Fin de programa");
13     }
14
15 }

```

Ilustración 68. Segundo if

```
la condición es verdadera
es un cinco
el doble de 5 es10
Fin de programa
Presione una tecla para continuar . . .
```

Ilustración 69. Ejecución segundo if

#### 8.1.1.1 Operadores de comparación

Los operadores que podemos usar dentro de los paréntesis de un *if* para comprobar una condición son:

==	igual a
!=	no igual a o distinto
>	mayor que
>=	mayor o igual que
<	menor que
<=	menor o igual que

Tabla 10. Operadores de comparación

Fijaros que a diferencia de otros lenguajes el igual no es “=” sino “==” y distinto o no igual es “!=”.

Si la x es mayor o igual a 5 entonces entra, si no, no.

```
2  public class Pruebas {
3
4    public static void main (String args[]) {
5      int x=8;
6
7      if(x>=5){
8        System.out.println("la condición es verdadera");
9        System.out.println("es un cinco");
10       System.out.println("el doble de x es "+(x*2));
11     }
12     System.out.println("Fin de programa");
13   }
14 }
```

Ilustración 70. Tercer if

```
la condición es verdadera
es un cinco
el doble de x es 16
Fin de programa
Presione una tecla para continuar . . .
```

Ilustración 71. Ejecución tercer if

Si la x no es 5 o mayor que 5 no ejecutará el *if*.

El *if* tiene otro compañero de viaje para cuando queramos que se haga algo si la condición es falsa:

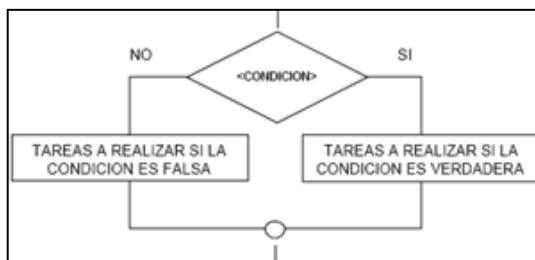


Ilustración 72. Estructura completa del if

En Java se utiliza la palabra reservada *else*:

```
2 public class Pruebas {
3
4     public static void main (String args[]) {
5         int x=8;
6
7         if(x>=5)
8             System.out.println("la condición es verdadera");
9         else
10            System.out.println("la condición es falsa");
11
12         System.out.println("Fin de programa");
13     }
14 }
15 }
```

Ilustración 73. Cuarto if

Si la condición es verdadera mostrará el primer mensaje, si no, el segundo. Siempre ejecutará el último (12) ya que se encuentra fuera del *if*. Un *if* puede tener o no *else* pero un *else* no puede existir sin un *if*. Si tenemos la necesidad de poner sólo un *else* es porque no hemos pensado bien la condición, hay que cambiarla para usarla con un *if*. Un *else* no lleva ninguna condición pues significa “si no”, la condición siempre va en el *if*.

Cuando usamos un *if-else* siempre se ejecutará uno de los dos, siempre, pero jamás los dos a la vez, o uno u otro.

Realmente en la condición de un *if* podemos poner cualquier cosa que como resultado pueda evaluarse a true o false:

```

3  public class Pruebas {
4
5      public static void main (String[] args) {
6
7          int x=9,y=5,z=25;
8          if(x*y>z)
9              System.out.println("dentro");
10
11     }
12 }
```

Ilustración 74. Quinto if

Cuando Java llega a la línea 8 primero hace la multiplicación y luego compara el resultado con z. En este caso como es verdad mostrará el mensaje.



Hay lenguajes como C en los que cualquier cosa distinta de 0 es verdad. Como ejemplo esta condición será válida if( $x+y$ ), de forma que siempre que la suma no sea 0 es verdad. En Java esto no se puede hacer y verdad es true y mentira false. Son tipos incompatibles con un entero.

#### 8.1.1.2 If's anidados

Como he comentado antes dentro de un *if-else* puede ir cualquier sentencia o sentencias compuestas por tanto dentro de un *if* pueden ir más *if*. A esto se le llama *if anidados*.

```

2  public class Pruebas {
3
4      public static void main (String args[]) {
5          int x=5;
6          int y=10;
7          int z=1;
8
9          if(x>=5)
10             if(y>8)
11                 if(z==0)
12                     System.out.println("todas las condiciones son verdaderas");
13                 else
14                     System.out.println("x e y son verdaderas pero z no");
15                 else
16                     System.out.println("x es verdad pero y no");
17             else
18                 if(x!=5)
19                     System.out.println("solo x es verdad y además no es 5");
20                 else
21                     System.out.println("solo x es verdad y además es 5");
22     }
23 }
```

Ilustración 75. If anidados

Veamos el resultado tal cual está:

```

x e y son verdaderas pero z no
Presione una tecla para continuar . . .
```

Ilustración 76. Resultado if anidados 1

Cambiemos y=2, z=0

```
x es verdad pero y no
Presione una tecla para continuar . . .
```

Ilustración 77. Resultado if anidados 2

Cambiemos x=3, y=5, z=0

```
solo x es verdad y además no es 5
Presione una tecla para continuar . . .
```

Ilustración 78. Resultado if anidados 3

Podéis usar llaves si lo veis más claro

Java sabe a qué *if* le corresponde un *else* por proximidad, se busca el *if* hacia arriba más próximo que no esté ya asociado.

**Importante:** Fijaros como el código no lo he escrito todo alineado en vertical. Cada vez que escribimos código que se encuentra dentro de un bloque hay que tabular hacia la derecha, dejar unos espacios, a esto se le llama identación. Si usáis llaves al pulsar *intro*, Geany automáticamente identa, si no lo tenemos que hacer nosotros. Hay lenguajes, como Python, que si el código no está bien identado no compilan. A java le da igual que este identado o no, como si queremos escribirlo todo en una línea, pero esto tiene un problema muy grande de legibilidad, el código no se puede entender.

Veamos el código anterior sin identar:

```
2  public class Pruebas {
3
4  public static void main (String args[]) {
5      int x=5;
6      int y=5;
7      int z=0;
8
9      if(x>=5)
10     if(y>8)
11     if(z==0)
12     System.out.println("todas las condiciones son verdaderas");
13     else
14     System.out.println("x e y son verdaderas pero z no");
15     else
16     if(x!=5)
17     System.out.println("solo x es verdad y además no es 5");
18     else
19     System.out.println("solo x es verdad y además es 5");
20   }
21 }
```

Ilustración 79. Ejemplo sin identación

Esto no lo entiende nadie. A mis alumnos les suelo decir: "no hay identación, no hay corrección". Por favor, **IDENTAD**.

### 8.1.1.3 Operadores lógicos

Para no tener que anidar muchos *if's* podemos usar ciertos operadores que me permiten unir distintas comparaciones en la condición del *if*. Dado que en el manual no vamos a realizar operaciones a nivel de bit voy a omitir estos y a ver los más usados.

Operador	Descripción	Sintaxis
<b>&amp;&amp;</b>	And lógico (y)	a && b
<b>  </b>	Or lógico (o)	a    b
<b>!</b>	Negación lógica (no)	!a

Tabla 11. Operadores lógicos

La mejor forma de ver el funcionamiento de estos operadores es mediante lo que se conoce como tablas de verdad<sup>6</sup>. Una tabla de verdad muestra la salida del operador ante las distintas posibilidades de entrada. Las tablas de verdad se suelen hacer solo con dos entradas, sabiendo ya como funcionan para dos se puede extraer para cualquier número de entradas.



Estos operadores los podéis encontrar en la bibliografía de algunos autores como “.”, multiplicación que se corresponde con el *and* y como “+”, suma que se corresponde con el *or*.

La V representa verdad y la F falso. Veamos las tablas de verdad, con todas las posibilidades de los operadores && y ||:

a	b	&&	
V	V	V	V
V	F	F	V
F	V	F	V
F	F	F	F

Tabla 12. Tabla de verdad AND y OR

a y b representan dos variables cuyo contenido puede ser verdadero o falso. Si analizamos la tabla de verdad de ambos podemos llegar a las siguientes conclusiones. && (and) solo es verdad cuando ambas variables lo son. || (or) sólo es falso cuando ambas variables lo son. Con esto podemos concluir que al *and* da verdadero sí y solo si todos sus operandos son verdaderos. Mientras que el *or* da verdadero sí y solo si un operando sea verdadero, solo con uno verdadero es suficiente, o lo que es lo mismo sólo puede dar falso si y solo si todos son falsos. Esto se cumple para cualquier número de operandos.

La negación en un operando muy simple: Niega lo que hay:

a	!a
V	F
F	V

Tabla 13. Tabla NOT

<sup>6</sup> Las tablas de verdad se usan mucho en electrónica digital donde también existen estos operadores llamándose puertas lógicas.

Si a es verdadero !a (no a) es falso. Aunque en principio puede parecer una tontería es un operando muy usado ya que nos permite hacer condiciones de una forma más simples y más cercanas a nuestra forma de razonar.

Existe un cuarto operador, no excesivamente usado que es el  $\wedge$ . Este operador es el XOR y su tabla de verdad es la siguiente:

a	b	$a \wedge b$
V	V	V
V	F	F
F	V	F
F	F	V

Tabla 14. Tabla XOR

Es verdad solo cuando uno de los operandos es verdad, solo uno.



No confundáis  $\wedge$  con exponente. En otros lenguajes es así pero Java no tiene un operador para ello. Veremos más adelante como se hace.

Veamos un ejemplo de estos operandos:

```

3 public class Pruebas {
4
5     public static void main (String[] args) {
6
7         int x=0,y=5,z=10;
8         if(x==0 && y>2)
9             System.out.println("primero");
10        if(x==8 || y>2)
11            System.out.println("segundo");
12        if(x==8 || y>10)
13            System.out.println("tercero");
14        if(x==0 && y>10)
15            System.out.println("cuarto");
16        if((x==0 && y>10)|| z<20)
17            System.out.println("quinto");
18
19    }
20 }
```

Ilustración 80. Operadores lógicos

```

C:\Windows\SYSTEM32\cmd.exe
primero
segundo
quinto
-----
(program exited with code: 0)
```

Ilustración 81. Salida operadores lógicos

Veamos lo sucedido:

1. **Se ejecuta la línea 8:**  $x==0$  es verdad,  $y>2$  es verdad, por tanto,  $V \&& V$  verdad.  
Entra en el *if* y muestra “primero”.
2. **Se ejecuta la línea 10:**  $x==8$  es mentira,  $y>2$  es verdad, por tanto,  $F \mid\mid V$  verdad.  
Entra en el *if* y muestra “segundo”.
3. **Se ejecuta la línea 12:**  $x==8$  es mentira,  $y>10$  es mentira, por tanto,  $F \&\& F$  falso.  
No entra en el *if*.
4. **Se ejecuta la línea 14:**  $x==0$  es verdad,  $y>10$  es mentira, por tanto,  $V \&\& F$  falso.  
No entra en el *if*.
5. **Se ejecuta la línea 16:**  $x==0$  es verdad,  $y>10$  es mentira, por tanto,  $(x==0 \&\& y>10)$  es falso.  $z < 10$  es verdad, falso (del anterior)  $\mid\mid$  verdad es verdad. Entra en el *if* y muestra “quinto”.

Ver anexo III



Fíjate en la prueba 3.  $F \&\& F$ . Ante este tipo de condiciones Java actúa de esta forma:  $x==8$  es falso ¿existe alguna posibilidad de que siendo la primera comparación falsa un *and* de verdadero? No. Un *and* ( $\&\&$ ) sólo puede dar verdad si todos lo son, por tanto, Java, viendo que el primero ya no es verdadero y está en un *and* no sigue evaluando el resto, por muchos operadores que hayan, da directamente falso. A esto se le llama evaluar en cortocircuito. Ahorra tiempo y nos proporciona una potente herramienta para hacer condicionales como veremos en próximos capítulos.

Veamos estas expresiones:

```

2 public class Pruebas {
3
4     public static void main (String[] args) {
5
6         boolean temp=true;
7         boolean temp2=false;
8
9         if(temp==true)
10            if(temp)
11
12             if(temp==false)
13                if(!temp)
14
15        }
16    }

```

Ilustración 82. Condiciones cortas

Las líneas 9 y 10 hacen exactamente lo mismo. Es lo mismo comparar con true que poner la variable sola, ya que si es verdadera es cuando entra.

Las líneas 12 y 13 hacen lo mismo. Es lo mismo comparar la variable con false que negarla ya que al escribir `!temp` le estamos diciendo *no temp* o sea falso, no entrará.

Existe un operador ternario que simula un *if* de forma corta y que personalmente no le tengo mucho aprecio pero por si lo veis que sepáis que es lo que hace:

```
condición ? si verdad : si falso
```

Ejemplo:

```
2 public class Pruebas {
3
4     public static void main (String[] args) {
5         int x=5;
6         int resultado;
7         resultado=(x==5) ? x+10 : x-10;
8         System.out.println(resultado);
9
10    }
11 }
```

Ilustración 83. Operador ternario

La línea 7 comprueba la condición entre paréntesis si es verdad realiza la primera operación antes del ':' si no la segunda. En ambos casos guarda el resultado en la otra variable.



Hemos visto como los operadores de comparación son para tipos primitivos, entonces ¿Por qué funciona el == con *String* si este tipo no es primitivo sino una clase? Como hemos visto los *String* (texto) son tan usados que Java permite un uso "cómodo" de los mismos de forma que se declaran y se utilizan como un tipo primitivo. Por otro lado, Java es muy listo y cuando ve que creamos dos variables con el mismo texto no crea dos espacios de memoria distintos para cada una, sino que ambas variables apuntan a la misma dirección de memoria.

Este modo de actuar puede causar algunos inconvenientes, como os explicaré en capítulos posteriores. Por ello para ver si dos objetos son iguales existe otra forma que nunca falla y es usar el método *equals*. *equals* (lo estudiaremos también) nos dice si dos *String* son iguales, por tanto, la forma correcta de ver si dos textos son iguales no es usando == sino *equals* y se usa de la siguiente forma: variable1.equals(variable2);

```
String texto1="hola";
String texto2="pepe";
if(texto1.equals(texto2))
.......
```

Como podéis observar una de las dos variables, da igual cuál, llama a *equals* y entre paréntesis se pone la otra. Antes del *equals* siempre va un '.' que indica que queremos usar ese método<sup>7</sup>.

Sé que esto es un poco complejo de entender ahora pero cuando veamos las clases lo entenderéis del todo. Vamos a hacer lo que yo llamo un salto de fe.

<sup>7</sup> Un pequeño programa para solucionar un problema que, en este caso, alguien ha hecho por nosotros.

### 8.1.2 switch

Cuando queremos hacer cosas distintas dependiendo del valor que tenga una variable podemos usar varios *if* comprobando el valor de la misma o usar la estructura *switch*.

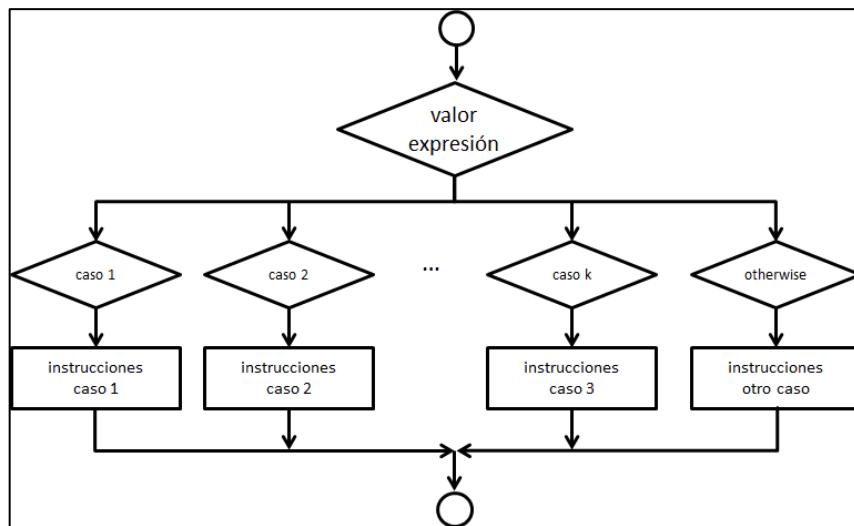


Ilustración 84. Estructura switch

Veamos como seria en Java mediante aproximaciones:

```

2 public class Pruebas {
3
4     public static void main (String[] args) {
5         int numero=3;
6
7         switch (numero){
8             case 1:System.out.println("Uno");
9             case 2:System.out.println("Dos");
10            case 3:System.out.println("Tres");
11        }
12    }
13 }
14 }
```

Ilustración 85. switch aprox1

1. En la línea 5 declaramos la variable *numero* y le damos el valor 3. Dependiendo del valor de *numero* queremos que imprima por pantalla su equivalente en texto.
2. En la línea 7 mediante la palabra reservada *switch* le indicamos entre paréntesis la variable a inspeccionar y entre llaves ponemos los casos.
3. Los distintos casos se ponen usando la palabra *case* y el valor que se espera seguido de ":".
4. Después de los ":" vienen las instrucciones que queremos que se ejecuten.

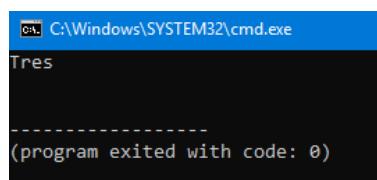


Ilustración 86. Salida switch aprox1

Bien, perfecto. Cambiemos el valor de *numero*:

```

2  public class Pruebas {
3
4      public static void main (String[] args) {
5          int numero=1;
6
7          switch (numero){
8              case 1:System.out.println("Uno");
9              case 2:System.out.println("Dos");
10             case 3:System.out.println("Tres");
11         }
12     }
13 }
14 }
```

Ilustración 87. switch aprox2

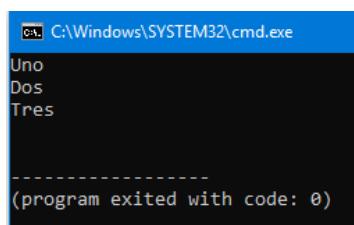


Ilustración 88. Salida switch aprox2

¿Qué ha pasado? Muy simple, el *switch* se ejecuta todo seguido entrando en todos los casos, para que esto no ocurra usamos la sentencia *break* que lo que hace es terminar el *switch* y seguir con la ejecución del programa. De esta forma poniendo un *break* en cada caso solo entrará en el correcto. El último *break* no es necesario pues después no hay más casos, yo personalmente siempre lo pongo.

```

2  public class Pruebas {
3
4      public static void main (String[] args) {
5          int numero=1;
6
7          switch (numero){
8              case 1:System.out.println("Uno");
9                  break;
10             case 2:System.out.println("Dos");
11                 break;
12             case 3:System.out.println("Tres");
13                 break;
14         }
15     }
16 }
17 }
```

Ilustración 89.switch aprox3

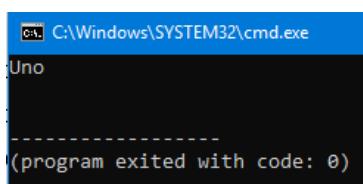


Ilustración 90. Salida switch aprox3

Ahora podemos ver como ya funciona correctamente. Dentro de un *case* podemos poner tantas sentencias como queramos entre sus ":" y el *break*. Existe una opción que se puede añadir en un *switch* para que se ejecute en caso de que ninguna de las opciones sea la correcta, *default*:

```

2  public class Pruebas {
3
4      public static void main (String[] args) {
5          int numero=8;
6
7          switch (numero){
8              case 1:System.out.println("Uno");
9                  break;
10             case 2:System.out.println("Dos");
11                 break;
12             case 3:System.out.println("Tres");
13                 break;
14             default:System.out.println("No es ninguno");
15         }
16     }
17 }
18 }
```

Ilustración 91. switch aprox4

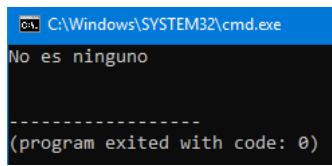


Ilustración 92. Salida switch aprox4

Como podéis ver al no existir el 8 entre los casos se ejecuta el por defecto.

Ejemplo con un tipo *char*:

```

2  public class Pruebas {
3
4      public static void main (String[] args) {
5          char letra='f';
6
7          switch (numero){
8              case 'a':System.out.println("Uno");
9                  break;
10             case 'd':System.out.println("Dos");
11                 break;
12             case 'f':System.out.println("Tres");
13                 break;
14             default:System.out.println("No es ninguno");
15         }
16     }
17 }
18 }
```

Ilustración 93. switch aprox5

En este caso como el tipo es *char* en los *case* se usan comillas simples.



Actualmente Java y el JDK 8 permite usar el *String* en un *switch* aunque no sea un dato de tipo primitivo.

## 8.2 Estructuras de repetición

Las estructuras de repetición o bucles se utilizan cuando queremos ejecutar sentencias cero o más veces. Tienen una condición de salida que nos indica cuando se deja de ejecutar el bucle y se continua con el programa.

### 8.2.1 while

El *while* es el primer bucle que tenemos y su sintaxis es la siguiente:

```
while (condición)
    <sentencias>
```

El *while* se repite mientras se cumpla la condición (sea verdadera). Lo que podemos poner entre paréntesis es exactamente lo mismo que lo visto en el *if* (operadores de comparación, lógicos y expresiones que den como resultado true o false) y al igual que este si hay más de una sentencia se pueden usar llaves.

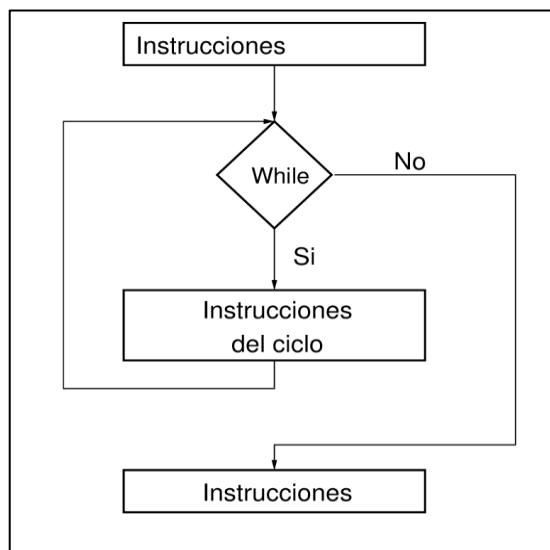


Ilustración 94. Estructura while

Veamos un ejemplo en el que vamos a mostrar los números del 0 al 9 por pantalla:

```

2 public class Pruebas {
3
4     public static void main (String[] args) {
5
6         int numero=0;
7
8         while(numero<10)
9             System.out.println(numero);
10
11     }
12 }
```

Ilustración 95. while aprox 1

En la línea 8 podemos ver como la condición del *while* es “mientras numero <10” de forma que hasta que *numero* no sea 10 no saldrá del bucle.

Veamos que sucede:

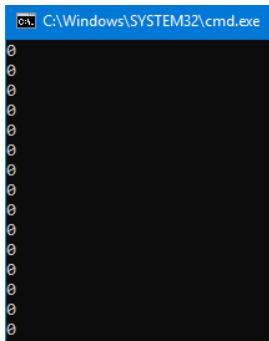


Ilustración 96. Salida while aprox 1

Solo hacen que salir ceros sin parar ¿Por qué? Porque no hay nada dentro del *while* que cambie el valor de la variable *numero* y por tanto la condición siempre es verdad.

```

2  public class Pruebas {
3
4      public static void main (String[] args) {
5
6          int numero=0;
7
8          while(numero<10){
9              System.out.println(numero);
10             numero=numero+1; //numero++
11         }
12     }
13 }
```

Ilustración 97. while aprox 2

Ahora después de mostrar el número se incrementa su valor en uno. Fijaros que lo podemos hacer como esta en la línea 10 o usando el operador incremento.

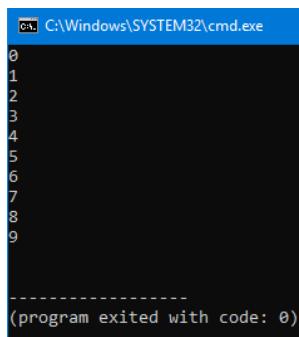


Ilustración 98. Salida while aprox 2

Ahora se va incrementando la variable y mostrándose hasta que llega un momento en que vale 10, no cumple la condición y el *while* termina.



Lo que ha sucedido en la primera aproximación se llama **bucle infinito**. Nunca salimos del *while* y el programa no funciona pudiendo llegar a consumir la memoria disponible. Para evitar esto debemos de asegurarnos que dentro del *while* hay alguna instrucción que en algún momento hará que la condición sea falsa.

### 8.2.2 do

La estructura *do* es la prima hermana del *while*, la única diferencia es que la condición se comprueba al final y no al principio.

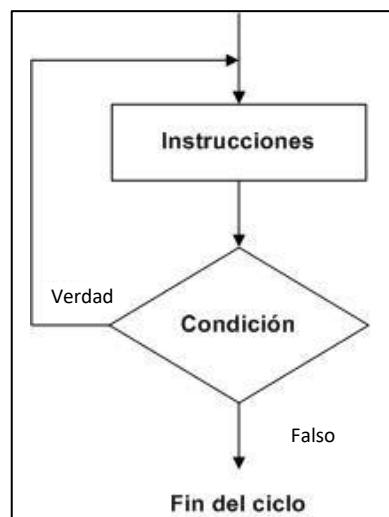


Ilustración 99. Estructura *do*

Fíjate que en la condición no es **mientras** se cumpla la condición sino **hasta** que se cumpla la condición, por tanto, se ejecuta el bucle si es falsa. La sintaxis es:

```

do{
    <sentencias>
}while(condición);
  
```

En este caso son necesarias las llaves siempre y como excepción a la regla después de la condición del *while* sí se pone “;”.

Hagamos el mismo ejemplo que con el *while* pero usando el *do*:

```

2 public class Pruebas {
3
4     public static void main (String[] args) {
5
6         int numero=0;
7
8         do{
9             System.out.println(numero);
10            numero=numero+1; //numero++
11        }while(numero<10);
12    }
13 }
  
```

Ilustración 100. *do*

La salida es exactamente la misma.

### 8.2.3 for

El *for* es un bucle un poco más complejo, veamos:

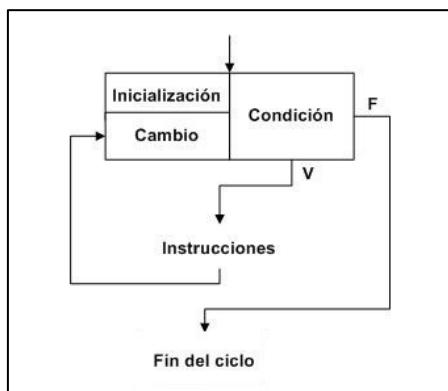


Ilustración 101. Estructura *for*

La sintaxis es la siguiente:

```
for(valor inicial;condición;incremento)
    <sentencias>
```

Las partes del *for* se separan por “;”.

1. Valor inicial: es el valor que tiene la variable que controla el *for*. Normalmente se suele declarar en el propio *for* y por algún motivo que desconozco suelen usarse las letras *i*, *j*, *t*.
2. Condición: nos indica cuantas veces se va a ejecutar el *for*. Para elegir una condición podemos pensar en “mientras...” como si fuera un *while*.
3. Incremento: es como va evolucionando la variable. A diferencia del *while* y el *do* la evolución de la variable la controla el propio *for*, por ello se suele llamar variable de control.

Veamos el mismo ejemplo de antes, pero hecho con un *for*:

```

2  public class Pruebas {
3
4      public static void main (String[] args) {
5
6          for(int i=0;i<10;i++)
7              System.out.println(i);
8
9      }
10 }
```

Ilustración 102. *for*

1. El resultado es exactamente el mismo. Veamos que ha sucedido:
2. En la línea 6 declaro la variable de control *i* y la inicializo a 0.
3. En la segunda parte digo que se ejecute el bucle mientras la *i* sea menor a 10.

4. En la última le digo que la *i* se tiene que incrementar de uno en uno. Es muy habitual usar los operadores unarios.

Al llegar a la línea 6 Java crea la variable y la inicializa, comprueba la condición y si es verdad entra. Después de una iteración<sup>8</sup> se produce el incremento y si se sigue cumpliendo la ejecución sigue iterando.

El *for* tiene muchas variaciones, por ejemplo, imaginad que quiero imprimir del 9 al 0, de mayor a menor:

```

2  public class Pruebas {
3
4      public static void main (String[] args) {
5
6          for(int i=9;i>=0;i--)
7              System.out.println(i);
8
9      }
10 }
```

Ilustración 103. *for* inverso

En este ejemplo se imprime por pantalla los números del 9 al 0. Fijaros en la inicialización de la variable, la condición y que se ha sustituido el incremento por un decremento.



Cada parte del *for* puede tener más de una expresión. Un *for* puede tener más de una variable, una condición con operadores lógicos y más de un incremento o decremento. Este *for* sería correcto *for(int i=0,j=5;i<10 && j>0;i++,j--)* si bien no es lo habitual.



En un bucle *for* la evolución de la variable la controla el mismo *for*. **Nunca modifiquéis la variable de control dentro del bucle.** Obtendréis resultados inesperados aparte de ser una muy mala praxis.

#### 8.2.4 ¿Cuál debo de usar?

A la pregunta cuál de las tres estructuras de repetición debo de usar la respuesta es fácil, pero con matices. En principio la que queráis, con la que vosotros os expreséis mejor o la condición la entendáis mejor. Pero hay que tener en cuenta una serie de cosas:

##### Diferencias entre *while* y *do*:

- En la estructura *while* primero se comprueba la condición y luego se ejecuta el bucle. Pudiera pasar que la condición sea falsa desde el principio y nunca se ejecuten las sentencias del bucle. El *while* se puede ejecutar 0 o muchas veces.

<sup>8</sup> Ejecución de las sentencias de un bucle

- En la estructura *do* la condición se comprueba al final, por tanto, aunque la condición sea falsa, la primera vez se va a ejecutar. El *do* siempre se ejecuta como mínimo una vez. Por tanto, se va a ejecutar 1 o muchas veces.

Un *while* se puede transformar en *do* y viceversa cambiando la condición y la posición del código.

#### Cuando usar while/do o for:

En un *for* siempre vamos a saber las veces que se va a ejecutar mientras que un *while/do* no. Siempre que no sepáis de antemano las veces que se va a repetir algo usad un *while* o *do* por ejemplo hasta que el usuario pulse una tecla (no sabemos cuándo lo va a hacer). Cuando sepamos las veces que se va a ejecutar (desde i hasta n) un *for* es más simple.

Un *for* también se puede cambiar por un *while* o *do*, siempre.



Cualquier combinación entre *if*, *switch*, *while*, *do* y *for* es posible.  
Podemos hacer que dentro de un *for* haya un *while* dentro dos *if* otro *for*...

Uso de char: Un *char* no es más que un entero que como ya hemos visto están codificados mediante la tabla ascii o Unicode por tanto puedo hacer esto sin tener que recurrir a métodos de transformación ni casting's

```

2 public class Pruebas {
3
4     public static void main (String[] args) {
5
6         char n='A';
7
8         while(n>='A' && n<='Z'){
9             System.out.print(n+" ");
10            n++;
11        }
12    }
13 }
```

Ilustración 104. Condición con char

```

C:\Windows\SYSTEM32\cmd.exe
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
-----
(program exited with code: 0)
```

Ilustración 105. Salida condición char

No, la Ñ no está codificada después de la N. Si miráis en la tabla del capítulo anterior veréis que es el número 165.

### 8.2.5 Break y continue

Estas son dos instrucciones que las comento porque aparecen en todas las bibliografías y debéis de saber lo que hacen. Bajo mi punto de vista estas instrucciones las carga el diablo y hacen más mal que bien.

- Break: el *break* es una instrucción que rompe la secuencia de ejecución natural del programa. Si ponéis un *break* dentro de un bucle, cuando se ejecute, Java saldrá automáticamente del mismo.
- Continue: hace exactamente lo mismo, pero en vez de salir del bucle fuerza la siguiente iteración.

```

2  public class Pruebas {
3
4  public static void main (String[] args) {
5
6      for(int i=0;i<10;i++)
7          if(i==6)
8              break;
9          else
10             System.out.println(i);
11
12 }
```

Ilustración 106. Break

Estas dos sentencias van en contra de la programación estructurada, rompen la continuidad del código y lo hacen ilegible. El ejemplo que os he puesto es muy simple, pero pensad en un programa con 500 líneas y 58 *break*, eso no lo entiende ni el que lo ha hecho.

Si usáis el *break*, lo más probable es que no habéis pensado bien la condición. Cualquier *break* se puede sustituir por una variable booleana para poder salir y cualquier *for* se puede transformar en un *while*.

Ya sé lo que va a pasar ahora, seguro que tenéis algún amigo que os dice que esto es una tontería y que no sé de lo que hablo, que con el *break* los programas van más rápidos... Me parece perfecto, para él todos, *arrieros somos y en el camino nos encontraremos*. Yo enseño programación no a codificar espaguetis.

¿Significa esto que jamás debemos de usar un *break*? **NO**. Obviamente como ya hemos visto el *break* es necesario en un *switch*. Pero no solo eso, cuando programéis de forma concurrente con threads y sockets pues sí, uno o dos *break* serán necesarios<sup>9</sup>.



No comencemos con malos hábitos, si queréis aprender bien no uséis el *break* ni el *continue*.

<sup>9</sup> Véase el servidor Apache

### 8.3 Ejercicios

1. Realiza un programa que realiza la siguiente figura por pantalla

```
*  
***  
*****
```

2. Realiza un programa que muestre por pantalla los 5 primeros números naturales.
3. Realizar un programa que muestre los 100 primeros números primos.
4. Realizar un programa que muestre las tablas de multiplicar.
5. Realizar un programa que muestre por pantalla los números del 1 al 100 sin mostrar los múltiplos de 5.
6. Realiza un programa que realiza la siguiente figura por pantalla.

```
*  
***  
*****  
*****  
***  
*
```

## 9. API

Como ya vimos Java es un lenguaje orientado a objetos puro. Nosotros estamos evitando el uso de la POO, pues la iremos viendo mediante aproximaciones, pero no podemos evitar tener que usar clases, objetos o métodos que ya están hechos. Java cuenta con una extensa librería, estructurada en paquetes, con un sinfín de clases y métodos que ya están hechos y podemos usar. Lo único que debemos de saber es cómo se usan.

Lo primero es acceder a esta API, para ello basta con buscar API Java 8 en google y es el primer resultado. No obstante os dejo el enlace a la API del JDK 8:

<https://docs.oracle.com/javase/8/docs/api/>

Nos encontramos con lo siguiente:

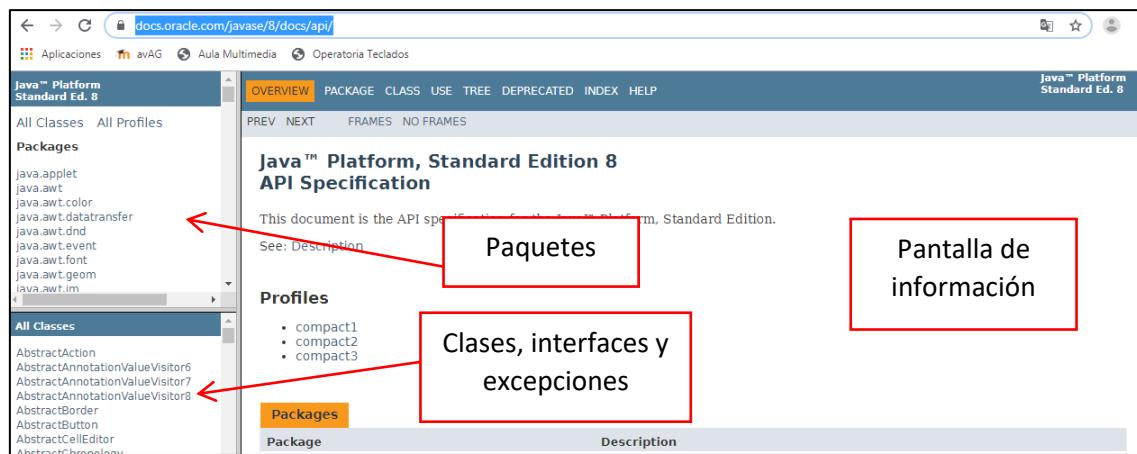


Ilustración 107. API 1

Veamos el ejemplo de la clase *String* que ya conocemos:

java.awt.image.renderable java.awt.print java.beans java.beans.beancontext java.io <b>java.lang</b> java.lang.annotation java.lang.instrument java.lang.invoke java.lang.management java.lang.ref java.lang.reflect java.math java.net java.nio	<b>int</b> <b>compareToIgnoreCase(String str)</b> Compares two strings lexicographically, ignoring case differences.  <b>String</b> <b>concat(String str)</b> Concatenates the specified string to the end of this string.  <b>boolean</b> <b>contains(CharSequence s)</b> Returns true if and only if this string contains the specified sequence of char values.  <b>boolean</b> <b>contentEquals(CharSequence cs)</b> Compares this string to the specified CharSequence.  <b>boolean</b> <b>contentEquals(StringBuffer sb)</b> Compares this string to the specified StringBuffer.  <b>static String</b> <b>copyValueOf(char[] data)</b> Equivalent to <code>valueOf(char[])</code> .  <b>static String</b> <b>copyValueOf(char[] data, int offset, int count)</b> Equivalent to <code>valueOf(char[], int, int)</code> .  <b>boolean</b> <b>endsWith(String suffix)</b> Tests if this string ends with the specified suffix.  <b>boolean</b> <b>equals(Object anObject)</b> Compares this string to the specified object.  <b>boolean</b> <b>equalsIgnoreCase(String anotherString)</b> Compares this String to another String, ignoring case considerations.  <b>static String</b> <b>format(Locale l, String format, Object... args)</b> Returns a formatted string using the specified locale, format string, and arguments.  <b>static String</b> <b>format(String format, Object... args)</b> Returns a formatted string using the specified format string and arguments.
---	---

Ilustración 108. API 2

Podemos ver como la clase *String* se encuentra en el paquete *java.lang* y tiene un método para comprobar si dos *String* son iguales que devuelve un booleano. Si pulsamos sobre el método nos da más información:

**equals**

```
public boolean equals(Object anObject)
```

Compares this string to the specified object. The result is true if and only if the argument is not null and is a String object that represents the same sequence of characters as this object.

**Overrides:**  
equals in class Object

**Parameters:**  
anObject - The object to compare this String against

**Returns:**  
true if the given object represents a String equivalent to this string, false otherwise

**See Also:**  
compareTo(String), equalsIgnoreCase(String)

Ilustración 109. API 3



Todo lo que hay en el paquete `java.lang` no es necesario importarlo a mano pues Java ya lo hace.

En el próximo tema vamos a tratar el tema de la entrada estándar, veamos donde se encuentran los métodos que vamos a estudiar:

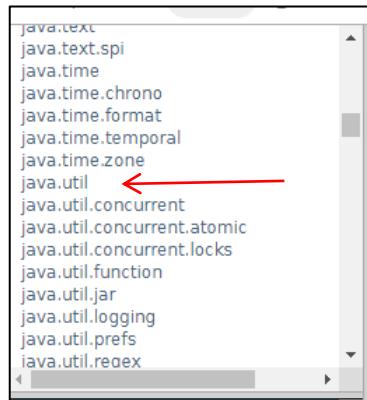


Ilustración 110. API 4

LongSummaryStatistics	nextInt()	Scans the next token of the input as a float.
Objects	nextInt()	Scans the next token of the input as an int.
Observable	nextInt(int radix)	Scans the next token of the input as an int.
Optional	nextLine()	Advances this scanner past the current line and returns the input that was skipped.
OptionalDouble	nextLong()	Scans the next token of the input as a long.
OptionalInt	nextLong(int radix)	Scans the next token of the input as a long.
OptionalLong	nextShort()	Scans the next token of the input as a short.
PriorityQueue	nextShort(int radix)	Scans the next token of the input as a short.
Properties	radix()	Returns this scanner's default radix.
PropertyPermission		
PropertyResourceBundle		
Random		
ResourceBundle		
ResourceBundle.Control		
Scanner		
ServiceLoader		
SimpleTimeZone		
Spliterators		
Spliterators.AbstractDoubleSpliterator		
Spliterators.AbstractIntSpliterator		
Spliterators.AbstractLongSpliterator		
Spliterators.AbstractSpliterator		
SplitterRandom		
Stack		
StringJoiner		
StringTokenizer		
Timer		
TimerTask		
TimeZone		
... More		

Ilustración 111. API 5

```
nextInt  
  
public int nextInt()  
  
Scans the next token of the input as an int.  
  
An invocation of this method of the form nextInt() behaves in exactly the same way as the invocation nextInt(radix), where radix is the default radix of this scanner.  
  
Returns:  
the int scanned from the input  
  
Throws:  
InputMismatchException - if the next token does not match the Integer regular expression, or is out of range  
NoSuchElementException - if input is exhausted  
IllegalStateException - if this scanner is closed
```

Ilustración 112. API 6

Scanner la veremos en el próximo capítulo y como podéis ver está en *java.util*. En definitiva, existen miles de “pequeños programitas” ya hechos para que los usemos.



Pese a la gran cantidad de librerías y soluciones que nos ofrece la API nosotros no vamos a usar todo su potencial ya que de lo que se trata en este manual es de aprender a programar no a usar una API. En próximos manuales veremos utilidades que nos facilitarán la vida.

## 10. Entrada por teclado

Hasta ahora hemos visto ejemplos en los que se les da un valor a unas variables y se resuelve un problema, pero esto dista mucho de la realidad. Como he comentado antes, tener una calculadora que solo pueda hacer operaciones sobre el número 5 y el 2 es totalmente inútil. Por tanto, debe de haber un mecanismo para que los programas sean más generales y reutilizables, ese mecanismo va a ser pedirle al usuario los valores con los que quiere operar.

La entrada por teclado se realiza mediante flujos de entrada llamados *streams* pero gracias a la API podemos poner una capa de abstracción y usar una clase que nos va a facilitar esta entrada. Esta clase es *Scanner*. *Scanner* se encuentra en el paquete *java.util*, por tanto lo primero que debemos de hacer antes de poder usarla es importarla. Podéis ver en la API los métodos que tiene, de todos ellos los que más vamos a usar son:

- *nextLine()*: Lee un texto.
- *nextInt()*: Lee un entero.
- *nextDouble()*: Lee un double.

(Obviamente hay muchos más, pero para los fines de este manual con estos nos sobran)

Lo primero que debemos de saber es que todo lo que se introduce por teclado es texto. Por tanto, si lo que se necesita es leer un entero lo que se hace es una conversión. Por ejemplo: si uso *nextLine()* como es texto no hace ninguna conversión ni nada, pero si lo que quiero es leer un entero al usar *nextInt()* lo que hace es leer como texto y luego transformarlo a entero.

Como he dicho antes para poder usar *Scanner* lo primero que debemos de hacer es importar la clase:

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

Esta línea se pone al principio de todo, antes de crear la clase principal. Con ella le decimos que importe la clase *Scanner* que se encuentra en el paquete útil dentro de las librerías de java. El camino hasta la clase en concreto se separa con puntos.

Una vez importada y ya dentro del programa debemos de crear un objeto (variable) que represente a *Scanner* y podamos usarlo por el nombre que le demos. *Scanner* necesita para poder crearse que le digamos de donde va a leer, si la salida estándar para imprimir es *System.out* la entrada estándar es *System.in*:

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

Con esta línea lo que creamos es una “variable” llamada *sc* del tipo *Scanner* y luego con el operador *new* le decimos que tiene que leer de la entrada estándar. “*sc*” es el nombre que le damos, podríamos llamarle *entrada* lo que sucede es que es común ponerle *sc* de esta forma al leerlo todos entienden que viene de un *Scanner*. En este manual y para la entrada estándar esta línea siempre será la misma. Una vez creada la “variable” ya podemos usar sus métodos.

Veamos un ejemplo:

```

1 import java.util.Scanner;
2 public class Pruebas {
3
4     public static void main (String args[]) {
5
6         Scanner sc=new Scanner(System.in);
7
8         String texto;
9         int numero;
10
11        System.out.print("Introduce un texto:");
12        texto=sc.nextLine();
13        System.out.print("Introduce un número:");
14        numero=sc.nextInt();
15        System.out.println();
16        System.out.println("El texto introducido es:"+texto);
17        System.out.println("El número introducido es:"+numero);
18
19    }
20 }
```

Ilustración 113. Prueba Scanner

1. En la línea 1 importamos Scanner.
2. Línea 6, dentro ya de donde estamos escribiendo el código lo primero que hacemos es crearnos una variable de tipo Scanner.
3. En las líneas 8 y 9 creo dos variables para recibir lo que el usuario va a introducir.
4. En la línea 11 muestro un texto para que el usuario sepa que debe de hacer, fijaros como uso *print* para que el cursor se quede esperando en la misma línea.
5. Después en la 12 llamo al método *nextLine()* del Scanner, para ello pongo el nombre de la variable (objeto), un punto y el método que quiero usar. El método es bloqueante, esto significa que Java se va a quedar esperando hasta que el usuario introduzca algo y pulse *intro*. Cuando el usuario pulsa *intro* coge lo que ha introducido y lo guarda en la variable texto.
6. Con las líneas 13 y 14 pasa lo mismo solo que *nextInt()* lee el texto lo transforma en un *int* y lo guarda en la variable número.
7. Las siguientes líneas muestran el contenido de las variables.

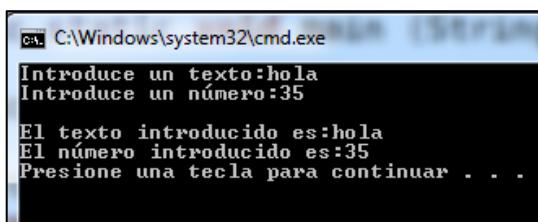


Ilustración 114. Salida Scanner

Ejemplo con un double:

```

1 import java.util.Scanner;
2 public class Pruebas {
3
4     public static void main (String args[]) {
5
6         Scanner sc=new Scanner(System.in);
7
8         double numero;
9
10        System.out.print("Introduce un número decimal: ");
11        numero=sc.nextDouble();
12        System.out.println();
13        System.out.println("El número introducido es:"+numero);
14
15    }
16

```

Ilustración 115. nextDouble()

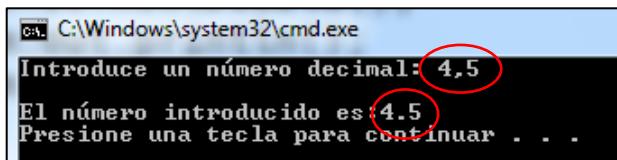


Ilustración 116. Salida nextDouble()

Fijaros en la salida. Para que Java entienda que es un decimal hay que usar la “,” no él “.”. Mientras que cuando lo muestra lo hace con “.”. Esto sucede porque Java usa el punto decimal pero como estás viendo estoy usando un entorno Windows y para este el decimal es la coma, por ello en la entrada debemos de teclear “,” ya que depende de Windows, no de Java.

¿Qué sucede si usamos el punto o peor todavía, si el usuario teclea algo que no es correcto?

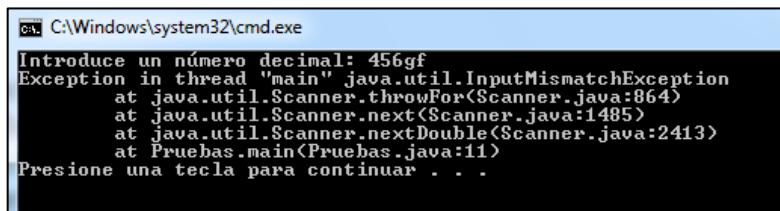


Ilustración 117. Salida excepción

Lo que se ha producido es una excepción, `nextDouble()` no ha podido transformar a `double` lo que hemos introducido ya que no es un número. Cuando salta una excepción es porque el programa no puede continuar. En otro capítulo explicaré, brevemente, lo que son las excepciones, no obstante con lo que hemos aprendido hasta ahora debemos de ser capaces de evitar este tipo de excepciones, tan solo hay que comprobar, antes de guardarla como `double`, que es un número, y para eso podemos usar las estructuras de control y repetición.



Cualquier next con conversión (nextInt, nextDouble...) lo que hacen es coger lo que hay que transformar, el número por ejemplo, y devolverlo en una variable pero dejan el intro que ha pulsado el usuario en el flujo de entrada. Cuando esto sucede si intentamos leer otra vez del teclado nos da la sensación de que Java pasa de nosotros y no pausa la ejecución para dejarnos escribir. Esto sucederá siempre que usemos un next con conversión y después uno sin. Por ejemplo: nextInt() y después un nextLine(). Solo sucede en este caso, no si primero hay nextLine() o si hay muchos nextInt() y nextDouble() pero ningún nextLine(). Cuando esto suceda lo que hay que hacer es antes de realizar el nextLine() limpiar el flujo de entrada y eliminar ese retorno de carro y lo más rápido es usar un nextLine() sin guardar su resultado. Por ejemplo:

```

10     System.out.print("Introduce un número decimal: ");
11     numero=sc.nextDouble();
12     sc.nextLine();
13     System.out.print("Introduce un texto: ");
14     texto=sc.nextLine()

```

La línea 12 lee el retorno de carro que ha dejado la 11 y lo tira a la basura ya que no lo guarda. Cuando se ejecuta la 14 esta funciona bien.

**Probad en casa lo que pasa cuando no ponéis la 12 y la solución.**

Métodos de la clase *String* que nos pueden ser de utilidad y sí podemos usar:

- charAt(int i): devuelve el carácter cuya posición es *i* dentro de un texto.
- length(): me dice la longitud de un texto.
- substring(int inicio, int fin): extrae una subcadena de un texto.

Todos los caracteres de un String tienen un índice que comienza por 0. Por ejemplo: "hola" tiene una longitud de 4 y sus índices son:

0	1	2	3
h	o	l	a

Por tanto, para recorrer un *String* solo podemos llegar hasta longitud-1.

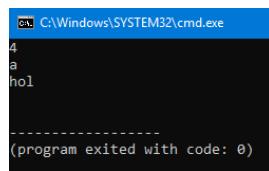
Esto lo entenderemos mejor en el capítulo de *String* y de vectores. Veamos un ejemplo de estos métodos:

```

3  public class Pruebas {
4
5      public static void main (String[] args) {
6
7          String texto="hola";
8
9          System.out.println(texto.length());
10         System.out.println(texto.charAt(3));
11         System.out.println(texto.substring(0,3));
12
13     }
14

```

Ilustración 118. *String*



A screenshot of a Windows Command Prompt window titled 'cmd.exe'. The window shows the output of a Java program. The text '4' is on line 4, 'a' is on line 5, and 'hol' is on line 6. Below the text, it says '(program exited with code: 0)'.

Ilustración 119. Salida String

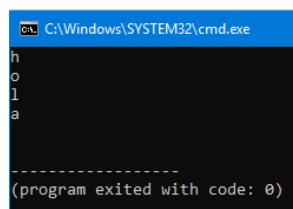
1. En la línea 9 preguntamos cuál es la longitud del *String*. (acordaros que los métodos siempre van entre paréntesis aunque no necesiten nada). El resultado es 4.
2. En la línea 10 mostramos el carácter que se encuentra en la posición 3. Teniendo en cuenta que se comienza a contar por 0 el carácter es la 'a'.
3. En la línea 11 sacamos una subcadena que va desde el carácter 0 al 2. Substring saca desde el índice inicial hasta uno menos del que pongamos como final (3). En este caso desde el 0 al 2 "hol".

¿Cómo sacar letra a letra de un texto?

```

3  public class Pruebas {
4
5     public static void main (String[] args) {
6
7         String texto="hola";
8
9         for(int i=0;i<texto.length();i++)
10            System.out.println(texto.charAt(i));
11
12    }
13
14 }
```

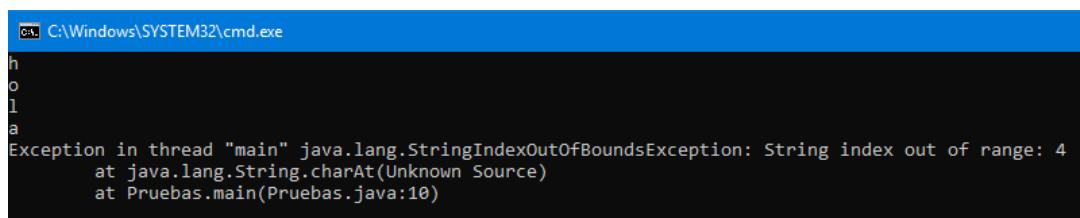
Ilustración 120. For String



A screenshot of a Windows Command Prompt window titled 'cmd.exe'. The window shows the output of a Java program. The letters 'h', 'o', 'l', and 'a' are printed one by one on separate lines. Below the text, it says '(program exited with code: 0)'.

Ilustración 121. Salida for String

En la línea 9 creamos un for que parte de 0 sin llegar hasta el final y que se incremente en uno. Con charAt(i) vamos sacando el *char* de cada índice que se corresponde con la *i*. Si la palabra es hola la *i* tomará los valores desde 0 hasta 3. ¿Qué sucede si en vez de <text.length() pongo <=text.length(). Estaré intentando acceder al índice 4, que no existe y me dará una excepción:



A screenshot of a Windows Command Prompt window titled 'cmd.exe'. The window shows the output of a Java program. It prints the letters 'h', 'o', 'l', and 'a' on separate lines. Then it prints an error message: 'Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String index out of range: 4' followed by the stack trace 'at java.lang.String.charAt(Unknown Source)' and 'at Pruebas.main(Pruebas.java:10)'.

Ilustración 122. Salida 2 for string

Al intentar acceder al 4 no puede y me dice que me he salido del rango permitido.



Cuidado, hemos visto nextLine() y sí, existe un next() pero no hace lo mismo. Leed bien la API.

## 10.1 Ejercicios

1. Realizar una calculadora básica de 4 operaciones (+, -, \*, /) con menú. Los resultados deben de ser congruentes.
2. Realizar los cambios pertinentes a la calculadora para que controle los números de opciones válidos y que se ejecute preguntando una opción hasta seleccionar 5-salir.
3. Mostrar un texto introducido por teclado del revés
4. Decir si un texto introducido esta en mayúsculas o minúsculas
5. La conjetura de Collatz afirma que, al partir desde cualquier número, la secuencia siempre llegará a 1. A pesar de ser una afirmación a simple vista muy simple, no se ha podido demostrar si es cierta o no.
6. Realizar un programa que dado un número entero introducido por teclado muestre su secuencia de Collatz. Siempre empieza en n y termina en 1

$$f(n) = \begin{cases} \frac{n}{2}, & \text{si } n \text{ es par} \\ 3n + 1, & \text{si } n \text{ es impar} \end{cases}$$

## 11. Vectores

Hasta ahora hemos usado variables para guardar nuestros datos, pero imaginaros que necesito guardar 100 enteros, crear una variable por cada uno no es inviable. Para esto existen los vectores (arrays, matrices, arreglos...). Un vector es una estructura a la que se le da un tamaño y un tipo y puede guardar tantos datos de ese tipo como su tamaño.



Los vectores son muy útiles en programación y todo el mundo debe de saber usarlos. Sí, sé que en Java existe algo llamado `ArrayList` y muchas cosas más, pero como siempre digo mi misión es enseñar a programar, no a codificar copiando una API. *Una vez sepamos caminar iremos en bicicleta.* PD: Mucha gente usa la clase `ArrayList` provocando un coste mayor que el uso de vectores.



Los vectores son estructuras estáticas, una vez definido su tamaño no se puede cambiar. Todos los datos guardados en un vector deben de ser del mismo tipo, no se pueden mezclar. (Más adelante veremos cómo esto es modulable).

La mejor forma gráfica de representar un vector es mediante un rectángulo dividido en casillas. Por ejemplo, un vector de 10 elementos con los números del 1 al 10 sería así:

índices →	0	1	2	3	4	5	6	7	8	9
valores →	1	2	3	4	5	6	7	8	9	10

Cada casilla del vector tiene un índice que es un entero y que comienza por 0 (similar a los String). Podemos acceder a una casilla mediante su índice. Dado que los índices comienzan por 0 si el vector es de 10 solo podemos llegar hasta el 9, si no nos salimos del vector y da una excepción. Para saber la longitud de un vector podemos usar el nombre del vector seguido de `.length`.



Este `length` **NO** lleva paréntesis ya que no es un método

Para declarar un vector y acceder usamos los corchetes. Veamos varios ejemplos y lo entenderemos mejor.

```

2  public class Pruebas {
3
4      public static void main (String[] args) {
5
6          int [] vec=new int[5];
7          int num;
8          vec[0]=2;
9          vec[1]=6;
10         vec[2]=4;
11         vec[3]=3;
12         vec[4]=8;
13
14         System.out.println(vec[3]);
15         num=vec[2];
16         System.out.println(num);
17
18     }
19 }
```

Ilustración 123. Vector 1

1. En la línea 6 declaramos el vector, para ello ponemos primero el tipo del que va a ser (todo el vector será de tipo *int*), luego para indicar que lo que estamos declarando es un vector lo indicamos con corchetes, después viene el nombre del vector. Una vez declarado le indicamos el tamaño usando el operador *new* seguido del tipo y entre corchetes el tamaño. Hemos declarado un vector de enteros de tamaño 5.
2. Para guardar un número en una casilla ponemos el nombre del vector y entre corchetes el índice de la casilla donde queremos guardar el número y le asignamos el valor (líneas de la 8 a la 11).
3. Para mostrar el contenido de la casilla de un vector podemos poner el nombre y el índice de la casilla en un *println*.
4. También podemos sacar el contenido y guardarlo en una variable (*num*).
5. El hecho de guardar en una variable o mostrar el contenido de un vector NO elimina lo que se encuentra en la casilla. El vector solo se ve afectado si está a la izquierda de una asignación.

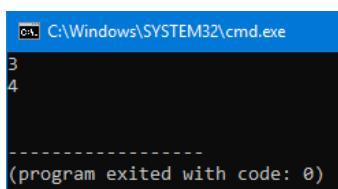


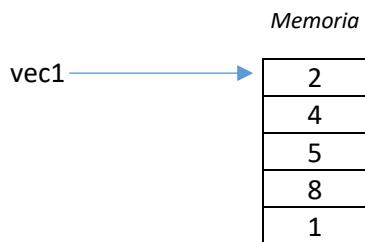
Ilustración 124. Salida vector 1



**Cuidado:** Java trata a los vectores como objetos si hacemos *vector1=vector2* no estamos haciendo una copia de *vector2* en *vector1*, *vector1* apunta a la misma dirección de memoria que *vector2*, son lo mismo. *vector1* es en realidad la dirección de memoria de la primera casilla que compone el vector.

Imaginemos el siguiente código:

```
int [] vec1=new int[5];
```

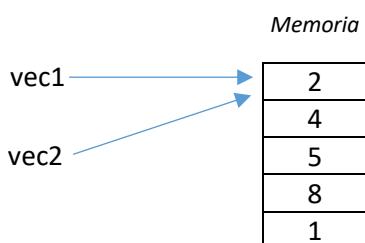


vec1 apunta a la dirección de memoria, vec1[2] es un entero que vale 5.

```
vec2[] int;
```

```
vec2=vec1;
```

El resultado es:



vec1 y vec2 son lo mismo, si uno cambia un numero se cambia para los dos:

```
vec2[2]=8;
```

dentro de vec1[2] también hay un 8

Muchas veces nos va a suceder que a la hora de declarar un vector no sabemos su tamaño porque depende de un cálculo o la entrada del usuario. Podemos declararlo arriba junto con las demás variables y darle el tamaño cuando lo sepamos:

```

1 import java.util.Scanner;
2 public class Pruebas {
3
4     public static void main (String[] args) {
5         Scanner sc=new Scanner(System.in);
6
7         int [] vec;
8         int n;
9
10        System.out.print("Introduce el tamaño del vector: ");
11        n=sc.nextInt();
12        vec=new int[n];
13
14    }
15 }
```

Ilustración 125. Vector 2

Fijaros como lo he declarado, le he pedido al usuario que introduzca el tamaño y luego lo he inicializado.

Otra forma de inicializar un vector es cuando sabemos de antemano los valores que va a tener:

```
int [] vec={2,5,3,7,8,4};
```

Se ponen los valores entre llaves separados por “,” sin usar el new. Java automáticamente crea el vector con el tamaño necesario y asigna a cada índice su valor.



A diferencia de las variables “normales” los vectores sí se inicializan de forma predeterminada. Si no les damos valor los enteros y decimales se inicializan a 0, los boolean a false, los char a nada y los vectores de objetos a null.

Obviamente llenar un vector o mostrarlo casilla a casilla como en el ejemplo 1 es algo arduo y tedioso. Para solucionar esto usamos las estructuras de repetición que hemos visto en capítulos anteriores. ¿Cuál creéis que es la más usada con vectores? Si habéis pensado en el *for* habéis acertado. Como vimos el *for* se usa cuando sabemos las veces que se va a repetir el bucle y en un vector lo sabemos. Además, la variable de control de un *for* se suele usar para conseguir el dato a guardar o algún resultado dado el vector.

Imaginad que queremos llenar un vector con los 10 primeros números naturales, luego multiplicar cada uno por 3 y terminar mostrando el resultado.



Para declarar un vector se puede usar la sintaxis C: int vec []. Donde el nombre va antes de los corchetes. Yo os recomiendo usar la propia de Java.

```
4 public class Pruebas {
5
6     public static void main (String[] args) {
7
8         int [] vec=new int[10];
9
10        for(int i=0;i<vec.length;i++)
11            vec[i]=i+1;
12
13        for(int i=0;i<vec.length;i++)
14            vec[i]=vec[i]*3;
15
16        for(int i=0;i<vec.length;i++)
17            System.out.print(vec[i]+" ");
18
19    }
20}
```

Ilustración 126. Vector 3

1. En la línea 8 declaro y le doy el tamaño al vector.

2. En el primer *for* recorro el vector. El primer índice de un vector es 0 y el último *length*, pero como yo quiero los números del 1 al 10 usando la propia *i* le sumo 1. Cuando la *i* sea 0 pondrá un 1, cuando sea 1 un 2 ...
3. Con el segundo *for* recorro el vector multiplicando lo que hay en cada casilla y guardándolo en la misma.
4. El último *for* imprime por pantalla en horizontal y separados por un espacio los números.

```
C:\Windows\SYSTEM32\cmd.exe
3 6 9 12 15 18 21 24 27 30
-----
(program exited with code: 0)
```

Ilustración 127. Salida vector 3



A la hora de aprender a usar vectores y poder rellenarlos con número nos puede ser de gran utilidad la clase *Random* que se encuentra en *java.util*. Esta clase tiene un método llamado *nextInt(int x)* que nos devuelve un valor aleatorio entre 0 y x-1.

Rellenemos y mostremos un vector de 10 elementos con número aleatorios entre 0 y 9:

```
3 import java.util.Random;
4 public class Pruebas {
5
6     public static void main (String[] args) {
7         Random rnd=new Random();
8
9         int [] vec=new int[10];
10
11        for(int i=0;i<vec.length;i++)
12            vec[i]=rnd.nextInt(10);
13
14
15        for(int i=0;i<vec.length;i++)
16            System.out.print(vec[i]+ " ");
17
18    }
19 }
```

Ilustración 128. Vector 4

Como veis al igual que el Scanner hay que importarla, una vez importada la forma más fácil de crear un objeto “variable” para poder usarla es usar la sintaxis de la línea 7. Dentro del primer *for* para generar un número aleatorio entre 0 y 9 llamo a *nextInt(10)* ya que el número generado estará entre 0 y 10-1.

En esta ejecución podéis ver como se generan números distintos:

```
0 4 4 0 2 7 0 9 1 2
1 8 8 9 1 0 8 1 9 6
0 6 1 1 4 7 4 2 7 7
```

Ilustración 129. Salida vector 4

Cada fila se corresponde con una ejecución distinta.

¿Y si quiero números entre 15 y 36 por ejemplo?:  $36-15=21$  dado que saca un aleatorio menos  $21+1=22$ . Me explico:

```
for(int i=0;i<vec.length;i++)
    vec[i]=rnd.nextInt(22)+15;
```

Ilustración 130. Random

Veamos los valores máximos y mínimos: Cuando se genere un 0 le sumamos 15, ya tenemos el valor mínimo. Como hemos puesto 22 el máximo que puede generar es 21, cuando salga 21 si le sumamos 15 es igual a 36 ya tenemos el máximo. Números entre 15 y 36.

## 11.1 Ordenación

Una de las cosas más importantes que tienen que ver con vectores son los algoritmos de búsqueda y ordenación. Dado que este manual no pretende enseñar estructuras de datos y que Java ya tiene soluciones para esto no voy a explicarlas aquí, lo veré en el siguiente manual. No obstante, si es importante probar, como mínimo, un algoritmo de ordenación y fijaros que he dicho probar ya que este tipo de algoritmos ya están hechos y probados, simplemente hay que conocerlos, pero no es necesario sabérselos de memoria.

Veamos el algoritmo de ordenación más simple que podemos encontrar, el de la burbuja. El algoritmo de la burbuja sirve para ordenar cualquier tipo de vectores, si bien nosotros veremos el ejemplo con enteros. Este algoritmo lo que hace es ir desplazando los números mayores hacia la derecha dejando los menores en la izquierda, como si fueran burbujas que van subiendo. De ahí su nombre.

Vector original					
45	17	23	67	21	
Iteración 1:					
45	17	23	67	21	Se genera cambio
17	45	23	67	21	Se genera cambio
17	23	45	67	21	No hay cambio
17	23	45	67	21	Se genera cambio
17	23	45	21	67	Fin primera iteración

Ilustración 131. Algoritmo de la burbuja

Este sería un ejemplo de la primera iteración del algoritmo. El algoritmo seguirá recorriendo el vector hasta su tamaño total.

Ejemplo en Java:

```

2 public class Pruebas {
3
4     public static void main (String[] args) {
5         int [] vec={3,8,5,6,12,4,23,7,9,1};
6         int aux;
7         for (int i = 0; i < vec.length - 1; i++) {
8             for (int j = 0; j < vec.length - i - 1; j++) {
9                 if (vec[j + 1] < vec[j]) {
10                     aux = vec[j + 1];
11                     vec[j + 1] = vec[j];
12                     vec[j] = aux;
13                 }
14             }
15         }
16
17         for(int i=0;i<vec.length;i++)
18             System.out.print(vec[i]+ " ");
19     }
20 }
```

Si es mayor se intercambian con la ayuda de aux

Ilustración 132. Burbuja 1

Ilustración 133. Salida burbuja

Como podéis ver el algoritmo funciona pero se puede mejorar. El problema de este código es que, aunque llegue un momento en que el vector ya este ordenado el sigue iterando hasta que el primer *for* lo recorra todo. Para evitar esto podríamos hacer este cambio:

```

z=1;
while(z<vec.length && cambio) {
    cambio=false;
    for(int j=0;j<(vec.length-z);j++) {
        if(vec[j] > vec[j+1]){
            aux = vec[j];
            vec[j] = vec[j+1];
            vec[j+1] = aux;
            cambio=true;
        }
    }
    z++;
}
```

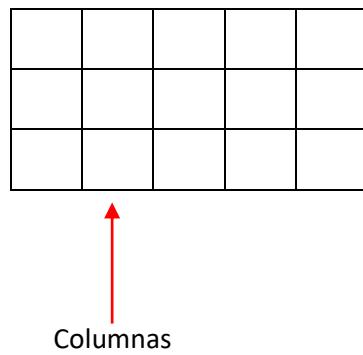
Ilustración 134. Burbuja 2

Añadimos una variable booleana que nos indica cuando hay un cambio, en el momento que sea *false* se sale de *while* sin terminar de recorrer lo que queda del vector.

## 11.2 Matrices

Los vectores pueden tener más de una dimensión, una matriz es un vector de dos dimensiones. Imaginad que dentro de una casilla de un vector creamos otro vector y lo representamos en columnas. Una matriz de 5x5 se vería así:


Filas →



Para posicionarnos en una casilla no es suficiente con una coordenada, necesitamos dos (fila, columna). Para declarar una matriz usamos dos corchetes, el primero indica las filas y el segundo las columnas. Para acceder a ella lo hacemos igual:

```
matriz[1][3]=7;
x=matriz[5][3];
```

Para recorrerlas necesitamos un *for* por cada dimensión:

```
3 import java.util.Random;
4 public class Pruebas {
5
6     public static void main (String[] args) {
7         Random rnd=new Random();
8         int [][] matriz=new int[10][10];
9
10    for(int i=0;i<matriz.length;i++){
11        for(int j=0;j<matriz.length;j++)
12            matriz[i][j]=rnd.nextInt(10);
13    }
14
15    for(int i=0;i<matriz.length;i++){
16        for(int j=0;j<matriz.length;j++)
17            System.out.print(matriz[i][j]+" ");
18        System.out.println();
19    }
20
21 }
22 }
```

Ilustración 135. Matriz 1

1. En la fila 8 podéis ver como se declara usando dos corchetes
2. En la 10 necesitamos dos *for* (¡¡OJO!! Las variables de control no pueden llamarse igual). Cuando el primer *for* tenga la *i=0* el segundo pasará por todas sus *j* (*j=0, j=1...*) y así sucesivamente de forma que por cada fila recorre todas las columnas.
3. Para mostrarla igual necesito los dos *for*, lo que hago es que para que salga por pantalla de forma “bonita” cada vez que termina el *for* de las columnas imprimo un retorno de carro.

```
C:\Windows\SYSTEM32\cmd.exe
3 9 1 0 4 8 8 4 9 6
1 8 9 2 0 3 1 8 4 6
0 9 8 4 6 4 9 2 4 4
2 8 5 5 4 7 6 7 7 8
0 1 1 0 2 1 8 6 9 3
5 9 3 7 2 0 6 6 2 7
6 5 9 1 1 0 5 8 6 2
1 0 8 0 4 4 1 0 9 8
1 2 1 4 5 6 7 7 3 4
8 3 6 2 4 0 7 5 4 0

-----
(program exited with code: 0)
```

Ilustración 136. Salida Matriz 1



Cuando necesitamos importar más de una clase de un paquete podemos usar el \* como comodín, que importa todas las clases de un paquete: `java.util.*`

Las matrices no tienen por qué ser cuadradas (filas=columnas) por ejemplo creamos una de 10x4:

```
3 import java.util.Random;
4 public class Pruebas {
5
6     public static void main (String[] args) {
7         Random rnd=new Random();
8         int [][] matriz=new int[10][4];
9
10    for(int i=0;i<matriz.length;i++){
11        for(int j=0;j<matriz.length;j++){
12            matriz[i][j]=rnd.nextInt(10);
13        }
14
15        for(int i=0;i<matriz.length;i++){
16            for(int j=0;j<matriz.length;j++)
17                System.out.print(matriz[i][j]+ " ");
18            System.out.println();
19        }
20    }
21 }
22 }
```

Ilustración 137. Matriz 2

```
C:\Windows\SYSTEM32\cmd.exe
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
at Pruebas.main(Pruebas.java:12)
```

Ilustración 138. Salida matriz 2

**¿What?** Algo ha ido mal. Si nos fijamos en el error la línea 12 está intentando acceder a al índice 4 pero ese índice no existe, el máximo es 3. Lo que está sucediendo es que en el segundo `for` hemos puesto `j<matriz.length` y `length` devuelve 10, siempre devuelve el tamaño de la primera dimensión. Como antes era cuadrada no ha pasado nada, pero ahora `j` no puede llegar a 10 solo a 3. Para especificar la longitud de la segunda dimensión lo que debemos es fijarnos en una fila de la primera, realmente cualquiera vale, pero lo más fácil es usar la 0. Para ello llamaremos a `length` pero de una fila en la primera dimensión:

```
3 import java.util.Random;
4 public class Pruebas {
5
6     public static void main (String[] args) {
7         Random rnd=new Random();
8         int [][] matriz=new int[10][4];
9
10    for(int i=0;i<matriz.length;i++){
11        for(int j=0;j<matriz[0].length;j++)
12            matriz[i][j]=rnd.nextInt(10);
13    }
14
15    for(int i=0;i<matriz.length;i++){
16        for(int j=0;j<matriz[0].length;j++)
17            System.out.print(matriz[i][j]+" ");
18        System.out.println();
19    }
20
21 }
22 }
```

Ilustración 139. Matriz 3

C:\Windows		
0	7	9 3
7	1	2 6
0	2	1 9
7	6	2 8
6	1	8 3
9	4	1 8
8	9	9 1
7	0	0 2
1	1	7 3
9	9	8 7

Ilustración 140. Salida matriz 3

Las matrices pueden tener las dimensiones que queramos, 3 sería un cubo. Cada dimensión que queramos añadir es un corchete más y un *for* más para recorrerla.



Aunque declaréis un vector o matriz y sepáis de antemano su tamaño no lo pongáis a piñón en los *for*, usad siempre *length*, de esta forma no os equivocareis sin querer y cambiando solo las dimensiones de la declaración todo seguirá funcionando.

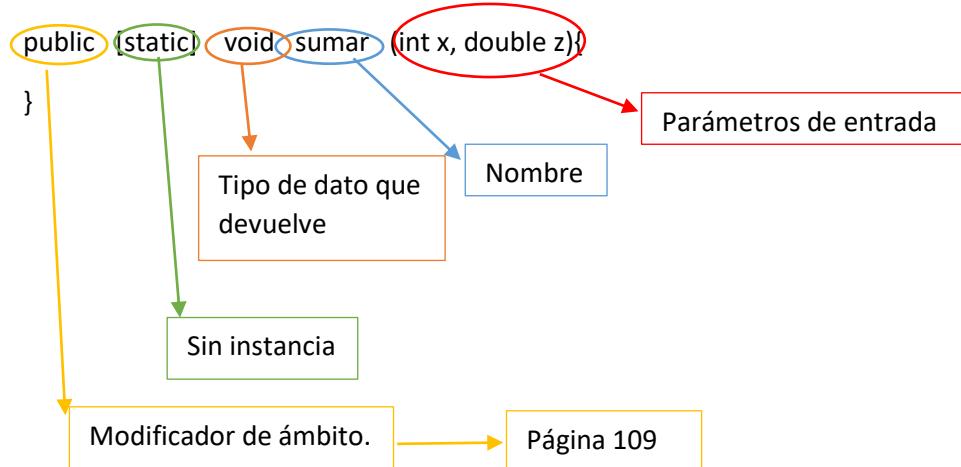
### 11.3 Ejercicios

1. Rellenar un vector con números del 0 al 9 y mostrarlo por pantalla.
2. Rellenar un vector con números del 1 al 10 y mostrarlo por pantalla.
3. Realizar un programa que dado un vector de 50 números aleatorios de 1 a 100 calcule la suma de todos y la media aritmética.
4. Hacer un programa que cree un vector de n elementos y lo rellene con números aleatorios del 1 al 100. Una vez lleno, mostrarlo por pantalla. Luego mostrar los números repetidos del vector junto a su número de apariciones. El número n lo pedirá por pantalla. Hay que controlar que el usuario solo pueda introducir números enteros, si no introduce un numero entero lo seguirá pidiendo.
5. Realizar un programa que mantenga una pequeña base de datos básica usando vectores sobre los alumnos matriculados y sus notas. El programa pedirá cuantos alumnos van a haber y luego pedirá el nombre y la nota de alumnos hasta introducir la letra 's'. Luego me permitirá consultar sus datos mediante el número de referencia asignado a los alumnos (índice) hasta introducir -1, con lo que terminará. Si se intentan matricular más alumnos de los permitidos el programa avisará de que el cupo ya está lleno. Hay que controlar que los índices para consultar sean válidos.
6. Crear y cargar dos matrices de tamaño 3x3 con números aleatorios. Mostrarlas. Sumarlas y mostrar su suma.
7. Generar una matriz cuadrada simétrica de 10x10 usando números aleatorios entre el 10 y el 99 y mostrarla por pantalla.
8. Una matriz cuadrada es simétrica si  $\forall i,j \in M \Rightarrow M(i,j) = M(j,i)$

## 12. Métodos

Hasta ahora hemos usado métodos ya definidos en alguna clase de la API de Java (`nextLine`, `println`...). Ahora vamos a aprender a crear nuestros propios métodos en la clase principal, junto al `main` para luego ya poder crearlos en las clases que queramos.

Los métodos se distinguen por un nombre, pueden tener uno, varios o ningún parámetro de entrada y devolver o no un resultado. También tienen un ámbito y siempre van entre llaves.



Ejemplos:

1. Método que muestra por pantalla "hola"

```
public static void mostrar() {
    System.out.println("hola");
}
```

2. Método que muestra por pantalla "hola" dato tu nombre

```
public static void mostrar(String nombre) {
    System.out.println("hola " + nombre);
}
```

3. Método que muestra por pantalla la suma de dos enteros

```
public static void sumar(int x, int y) {
    System.out.println("la suma es:" (x+y));
}
```

4. Método que calcula la suma de dos enteros y devuelve el resultado al programa principal

```
public static int sumar(int x, int y) {
    return x+y;
}
```

Cuando se llama a un método el programa se apunta la línea donde estaba y salta hasta la dirección de memoria donde está el método. Cuando termina de ejecutar el método se vuelve a la línea almacenada que es la siguiente a la de la llamada.

Imaginemos el ejemplo 1: Para llamar al método desde el programa principal (*main*) haríamos esto:

```
mostrar();
```

El programa salta hasta la dirección del método y lo ejecuta. Luego vuelve al *main*. Saldrá por pantalla: “*hola*”.

Los métodos pueden devolver un resultado al *main* después de ejecutarse. Ese resultado se puede guardar en el *main* en una variable.

Imaginemos el ejemplo 2: Para llamar al método desde el programa principal (*main*) haríamos algo así:

```
mostrar("pepe");
```

Se llama por su nombre y entre paréntesis se le pasa lo que necesita para realizar su tarea, en este caso un *String*. Fijaros como el siguiente código no tendría sentido:

```
int x;  
x=mostrar("pepe");
```

El método está declarado como *void*, por lo que no devuelve ningún resultado, lo que hace lo hace desde dentro sin devolver ningún dato al *main*. Mostraría por pantalla: “*Hola pepe*”. Sigamos y lo entenderéis.

## 12.1 Static

Cuando declaramos un método como *static* en una clase significa que ese método va a ser el mismo para todos los objetos que se creen. Esto es un poco complejo de explicar antes de ver como declarar nuestras propias clases, de momento como vamos a crear todos los métodos en la misma clase donde se encuentra el *main*, los declararemos como *static*. Cuando lleguemos al capítulo de las clases lo entenderéis fácilmente. Se podría decir que se encuentra en un entorno estático, no instanciable (no se necesita crear un objeto para usarlo).

## 12.2 Modificador de ámbito

Los modificadores de ámbito me permiten definir desde donde puede ser o no accedido un método. Nos podemos encontrar con *public* (cualquier sitio), es el que usaremos nosotros de momento, *protected* y *private*. Estos modificadores se verán con más detalle en próximos capítulos<sup>10</sup>. Todos los métodos que usemos en este capítulo serán *public static* ya que se encuentran en la misma clase que el *main*.

---

<sup>10</sup> Página 109

### 12.3 Return

Veamos ahora el ejemplo 4: En el *main* el código, por ejemplo, sería este:

```
int x;  
x=sumar(5,2);
```

La variable *x* acaba teniendo un 7. Veamos los pasos:

- Declaro la variable *x*.
- Java ve que la segunda línea es una asignación, con lo que primero tiene que resolver la parte de la derecha, se da cuenta que es un método y lo llama pasándole los valores 5 y 2, ya que son los parámetros que necesita el método para funcionar (dos parámetros de tipo *int*).
- Java se va al código donde se encuentra el método *sumar* y lo ejecuta. Se encuentra con un *return* de una suma. Calcula la suma y sobre el resultado ejecuta el *return*. El *return* podríamos decir que asigna al nombre del método el resultado (7) y devuelve la ejecución al *main*.
- El *main* ya sabe que la parte de la derecha de la asignación vale 7 y lo guarda en la *x*.

Al final la *x* vale 7.



Como este método va a devolver un entero, va a tener un *return* por lo que no ponemos *void* sino el tipo que va a devolver: *int*   
*public static int* ... El *void* se usa cuando un método no devuelve nada, no hay *return*.

Un *return* dentro de un método siempre para la ejecución del mismo y devuelve la ejecución al método que lo llamó, por ejemplo, el *main*. El uso del *return* de forma indiscriminada provoca los mismos problemas que un *break* ¿eso significa que no puedo poner más de uno?, no, si el código es simple y se entiende enseguida no pasa nada por poner más de uno, por ejemplo:

```
if(x>0)  
    return true;  
else  
    return false;
```

En este código solo se ejecutará un *return*, dependiendo de si entra en el *if* o en el *else* y está muy claro, pero no abuséis de esto, usad variables temporales y poned un solo *return* al final del método.

## Ejemplos:

En el primer ejemplo el método está declarado como *void*, no devuelve nada, cuando termina sigue con la siguiente línea donde se ha quedado en el *main*. Se le llama por su nombre y no necesita nada:

```

2  public class Pruebas {
3
4      public static void menu(){
5          System.out.println("      Menú      ");
6          System.out.println("-----");
7          System.out.println("  1-Añadir");
8          System.out.println("  2-Modificar");
9          System.out.println("  3-Borrar");
10         System.out.println("  4-Salir");
11     }
12
13     public static void main (String args[]){
14         menu();
15         System.out.println("Fin de programa");
16     }
17 }
```

Ilustración 141. Ejemplo void

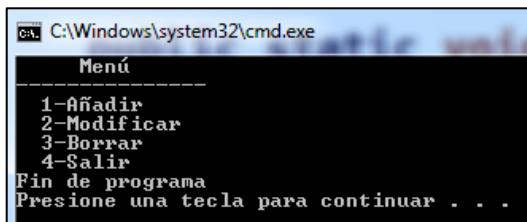


Ilustración 142. Salida void

En el segundo ejemplo vamos a realizar un método que calcule la media de dos números enteros. Devuelve un *double* que lo recoge el *main* y lo imprime. El método ya no es *void* y tiene un *return* indicando lo que va a devolver. También cuenta con dos parámetros que indican los números de los que se quiera sacar la media. Divido por 2.0 para que haga la división decimal:

```

2  public class Pruebas {
3
4      public static double media(int x, int y){
5          double resultado=(x+y)/2.0;
6          return resultado;
7      }
8
9      public static void main (String args[]){
10         double operacion;
11         operacion=media(3,8);
12         System.out.println(operacion);
13         System.out.println("Fin de programa");
14     }
15 }
```

Ilustración 143. Return

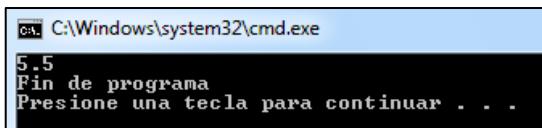


Ilustración 144. Salida Return

## 12.4 Paso de parámetros

Cuando hablamos de parámetros nos referimos a los datos que se le hacen llegar a un método para que pueda realizar su cometido.

Nos encontramos con dos tipos, los parámetros formales y los parámetros reales:

- **Parámetros formales:** Son las variables que recibe el método, se crean al definir el mismo, son los que se ponen entre paréntesis y se usan como variables locales dentro del método. Su contenido lo recibe al realizar la llamada al método de los parámetros reales. Cuando el método termina su ejecución, mueren.
- **Parámetros reales:** Son las expresiones que se utilizan en la llamada al método, sus valores se copiarán en los parámetros formales.

Ejemplo:

```

2  public class Prueba {
3
4      public static int sumar(int x,int y){
5          int r;
6          r=x+y;
7          return r;
8      }
9
10     public static void main (String[] args) {
11
12         int numero1=5;
13         int numero2=6;
14         int resultado;
15
16         resultado=sumar(numero1,numero2);
17     }
18 }
```

Ilustración 145. Parámetros formales y reales



El nombre de los parámetros formales y reales no tienen por qué ser distintos. En el caso anterior a los formales les podríamos haber llamado *numero1* y *numero2* haciendo *r=numero1+numero2* y hubiera funcionado igual. Una cosa es el nombre dentro del método y otra fuera. Si los ponemos igual Java no se va a liar. Si sois novatos os recomiendo ponerlos distintos hasta que os hagáis a la programación.

Los parámetros en java se pueden pasar de dos formas:

- **Por valor:** se le pasa al método una copia del valor de la variable, de esta forma si el parámetro sufre algún cambio dentro del método no se ve reflejado en la variable del llamante.

```

4  public class Prueba {
5
6      public static int sumar(int x, int y){
7          int z=x+y;
8          x=100;
9          return z;
10     }
11     public static void main (String[] args) {
12         int x=5;
13         int y=1;
14         int suma;
15         suma=sumar(x,y);
16         System.out.println("la suma es:"+suma);
17         System.out.println("la y vale:"+y);
18     }
19 }
20

```

Ilustración 146. Parámetros por valor

El resultado es:

Ilustración 147. Salida por valor

Veamos que ha ocurrido en la memoria principal: Primero se ejecuta el main, crea la variable x guardando un 5 en ella, luego crea la variable y guardando un 1 en ella.

Dirección	Nombre que le da	Contenido	Línea de código
0x00F2	x (main)	5	12
0x00F4	y (main)	1	13
0x00F6			
0x00F8			
0x00FA			
0x00FC			
0x00FE			

Tabla 15. Ejemplo parámetros por valor 1

Crea la variable suma sin inicializarla.

Dirección	Nombre que le da java	Contenido	Línea de código
0x00F2	x (main)	5	12
0x00F4	y (main)	1	13
0x00F6			
0x00F8			
0x00FA	suma	null	14
0x00FC			
0x00FE			

Tabla 16. Ejemplo parámetros por valor 2

Llama al método sumar pasándole una copia de los valores de x e y.

Dirección	Nombre que le da java	Contenido	Línea de código
0x00F2	x (main)	5	12
0x00F4	y (main)	1	13
0x00F6			
0x00F8			
0x00FA	suma	null	14
0x00FC	x de sumar	5	6
0x00FE	y de sumar	1	6

Tabla 17. Ejemplo parámetros por valor 3

Se crea z con el resultado de la suma.

Dirección	Nombre que le da java	Contenido	Línea de código
0x00F2	x (main)	5	12
0x00F4	y (main)	1	13
0x00F6			
0x00F8	z	6	7
0x00FA	suma	null	14
0x00FC	x de sumar	5	6
0x00FE	y de sumar	1	6

Tabla 18. Ejemplo parámetros por valor 4

Guarda un 100 en la x del método.

Dirección	Nombre que le da java	Contenido	Línea de código
0x00F2	x (main)	5	12
0x00F4	y (main)	1	13
0x00F6			
0x00F8	z	6	7
0x00FA	suma	Null	14
0x00FC	x de sumar	100	8
0x00FE	y de sumar	1	6

Tabla 19. Ejemplo parámetros por valor 5

El método devuelve *z*. Al terminar el método su *x*, *z* e *y* mueren. La *y* del *main* no se ha modificado porque se había pasado una copia de su valor, sigue valiendo 1. Al final suma vale 6.

Dirección	Nombre que le da java	Contenido	Línea de código
0x00F2	x (main)	5	12
0x00F4	y (main)	1	13
0x00F6			
0x00F8			
0x00FA	suma	6	15
0x00FC			
0x00FE			

Tabla 20. Ejemplo parámetros por valor 6

- **Por referencia:** al método se le pasa un puntero a la memoria donde se guarda la variable, vamos, la dirección de memoria de la variable original. Veamos: En el siguiente ejemplo creo un vector de tres posiciones con los números 1, 2 y 3. Lo muestro y luego llamo al método para intentar cambiar el número de la posición 1.

```

4  public class Prueba {
5
6      public static void cambiar(int [] vector){
7
8          vector[1]=87;
9      }
10     public static void main (String[] args) {
11         int [] vec={1,2,3};
12
13         for(int i=0;i<vec.length;i++)
14             System.out.print(vec[i]+" ");
15
16         System.out.println();
17         System.out.println("-----");
18         cambiar(vec);
19
20         for(int i=0;i<vec.length;i++)
21             System.out.print(vec[i]+" ");
22
23
24     }
25
26 }
```

Ilustración 148. Parámetros por referencia

Salida por pantalla:

```

1 2 3
-----
1 87 3

```

Ilustración 149. Salida Parámetros por referencia

Veamos que ha ocurrido en la memoria principal: Se crea el vector *vec*

Dirección	Nombre que le da java	Contenido	Línea de código
0x00F2	vec[0](main)	1	11
0x00F4	vec[1](main)	2	11
0x00F6	vec[2](main)	3	11
0x00F8			
0x00FA			
0x00FC			
0x00FE			

Tabla 21. Ejemplo parámetros por referencia 1

Como Java ha pasado la variable *vec* por referencia, *vec* y *vector* son lo mismo, no es una copia, sino que apuntan a la misma dirección de memoria.

Dirección	Nombre que le da java	Contenido	Línea de código
0x00F2	vec[0](main)	1	11
0x00F4	vec[1](main)	2	11
0x00F6	vec[2](main)	3	11
0x00F8			
0x00FA	vector en cambiar	0x00F2	6
0x00FC			
0x00FE			

Tabla 22. Ejemplo parámetros por referencia 2

Cambio el contenido de la posición 1 del *vector*, pero como son el mismo se cambia en el original.

Dirección	Nombre que le da java	Contenido	Línea de código
0x00F2	vec[0](main)	1	8
0x00F4	vec[1](main)	87	11
0x00F6	vec[2](main)	3	11
0x00F8			
0x00FA	vector en cambiar	0x00F2	6
0x00FC			
0x00FE			

Tabla 23. Ejemplo parámetros por referencia 3

Cuando termina el método y muestra el vector lo hace con los cambios.

Dirección	Nombre que le da java	Contenido	Línea de código
0x00F2	vec[0](main)	1	8
0x00F4	vec[1](main)	87	11
0x00F6	vec[2](main)	3	11
0x00F8			
0x00FA			
0x00FC			
0x00FE			

Tabla 24. Ejemplo parámetros por referencia 4



El acceso a una casilla del vector se hace sumando lo que ocupa cada casilla a la dirección de la primera que se guarda en memoria.

## 12.5 Recursividad

La recursividad es una propiedad que tienen los métodos (funciones en otros lenguajes) de llamarse a sí mismas para resolver un problema. En el cuerpo del método se encuentra una sentencia que es una llamada al propio método. Veamos un ejemplo y su funcionamiento:

El factorial de un número es un problema por sí mismo recursivo:

$$n! = n * (n-1)!$$

El factorial de 4 es:  $4! = 4 * 3 * 2 * 1$  (no pongo el 0 cuyo factorial es 1). De forma que el factorial es el producto del número por todos los anteriores hasta uno, por tanto podríamos pensar que para calcular el factorial de  $n$  llamamos al método y desde el cuerpo de este se va llamando a si mismo tantas veces como necesita con  $n-1$ . Obviamente esto tiene un peligro y es crear un bucle infinito que produzca que no se salga nunca del método, incluso que acabemos con la memoria disponible, para que esto no suceda debemos de asegurarnos de que en un momento las llamadas terminarán, en este caso cuando  $n$  sea 1.

Las consecutivas llamadas al método se van apilando en la memoria, pues el método no sabe el factorial de 4, 3 o 2, pero llega un momento en que se pide el factorial de 1 y ese sí sabe resolverlo: `return 1`. Cuando esto sucede ya puede ir calculando el factorial de 2, 3 y 4 desapilando de la memoria y liberándola.

```

2  public class Prueba {
3
4      public static int factorial(int n){
5          if(n == 1)
6              return 1;
7          else
8              return n * factorial(n-1);
9
10     }

```

Ilustración 150. Recursividad

Hagamos una traza, imaginemos  $n=4$

1. Cuando entra en la función ve que  $n$  es distinto de 1 y ejecuta el `else` llamando al propio método, posponiendo la decisión de calcular el factorial de 4 y apilando la llamada a factorial de  $(n-1)$ .
2. La segunda vez que entra  $n=3$ , tampoco puede resolverlo, apila factorial de  $(n-1)$ .
3. La tercera vez  $n=2$ , tampoco puede resolverlo, apila factorial de  $(n-1)$ .
4. La cuarta vez  $n=1$  y esto sí sabe resolverlo ya que si `if(n==1) return 1`, en otras palabras le hemos dado



- una salida y es que cuando n es igual a 1 debe de devolver 1.
5. Ahora comienza a desapilar y calcular factorial(2)=2\*1.
  6. factorial(3)=3\*2
  7. factorial(4)=4\*6=24. En este punto termina y el método nos puede dar el resultado: 24



#### Cosas a tener en cuenta de la recursividad:

- Tiene un mayor coste, tanto en tiempo de ejecución como en memoria.
- Es más limpia y legible.
- Hay problemas que tienen una solución propiamente recursiva (factorial, torres de Hanoi...).
- Siempre será más rápido y tendrá un menor coste una solución iterativa.

Recordemos con lo aprendido en capítulos anteriores como sería el ejemplo del factorial de forma iterativa y por ende menos costosa.

```

4   public static int factorial(int n){
5       int fac=1;
6       for(int i=2;i<=n;i++)
7           fac=fac*i;
8       return fac;
9   }

```

*Ilustración 151. Factorial iterativo*

#### Cosas a tener en cuenta sobre los métodos:

- A la línea que define el método se le llama cabecera. El código entre llaves es el cuerpo.
- En la cabecera ponemos *static* cuando se encuentran en una clase que no se va a instanciar (de momento siempre).
- Ponemos *void* cuando no queremos que devuelvan ningún resultado.
- Si devuelven algo siempre ponemos el tipo que devuelven en la cabecera y se usa la palabra reservada *return* en el cuerpo.
- Las nombramos con nombres significativos.
- Si el nombre está compuesto por más de una palabra escribimos todo en minúscula excepto la primera letra a partir de la segunda palabra (sumarNumerosPares).
- Después del nombre siempre van paréntesis, aunque no se necesiten parámetros.
- Los parámetros son los valores que el método necesita para hacer su cometido.

- Se llaman parámetros formales a las variables declaradas entre paréntesis, se crean al definir la función. Su contenido lo recibe al realizar la llamada al método de los parámetros reales. Los parámetros formales son variables locales dentro de la función.
- Se llaman parámetros reales a las expresiones que se utilizan en la llamada al método, sus valores se copiarán en los parámetros formales.
- Los métodos deben de realizar cosas simples, no muchas cosas en uno solo.
- Por norma general, los métodos no deben de comunicarse con el exterior (teclado, pantalla), eso es problema del programa principal.

## 12.6 Ejercicios

1. Realizar un programa usando métodos que me diga si una frase introducida por teclado es un palíndromo.
2. Realizar un programa usando métodos que me diga la letra de un DNI.
3. Un programa con un método que reciba dos vectores de 10 elementos y devuelva uno de 20 ordenado ascendentemente. Se generarán con números aleatorios de 1 a 100. hay que mostrarlos todos.
4. Crear un programa para jugar al tres en raya. Dividir el juego en métodos por ejemplo: comprobarDiagonal, comprobarColumnas, tirada, etc.

## 13. Clases

En este capítulo vamos a “pegar” todas las piezas que hemos ido viendo y utilizar la orientación a objetos. Como comentaba al principio del manual, el mundo lo podemos ver como objetos, todo son objetos (coches, personas, lápices, ...), la programación orientada a objetos lo que trata es de subir un nivel más la abstracción de forma que programar se aproxime más a nuestra forma de interactuar en la vida real, dado que en la vida real tenemos una serie de objetos y los usamos, en Java también. Por ejemplo, en la vida real existe un objeto llamado coche que tiene unas propiedades (color, matrícula...) y una serie de acciones que se pueden realizar sobre él (conducir, repostar...), en programación orientada a objetos crearemos esos objetos y los usaremos.

Un objeto tiene una serie de características:

- **Identidad:** Cada objeto es único y diferente de otro. Yo puedo tener dos coches, pero no son el mismo, son dos coches distintos.
- **Estado:** Son las propiedades o atributos que tiene ese objeto (color, peso, matrícula...)
- **Comportamiento:** Las cosas que puede hacer, los métodos que tiene (repostar, conducir, aparcar...).
- **Mensajes:** Los objetos pueden interactuar entre ellos usando los métodos.
- **Métodos:** Los métodos propiamente dichos y que pueden realizar cálculos, acciones, cambiar las propiedades, etc.



String, Scanner, Random son clases que ya hemos usado, aunque no sepamos muy bien el funcionamiento. Aparte de entenderlas vamos a aprender a crear las nuestras propias. Haciendo las nuestras entenderemos la API.



Para explicar la orientación a objetos, como en otros apartados lo haré a través de ejemplos sucesivos de más simples a más complejos hasta llegar a verlo tal como debe de ser.

### 13.1 Clases vs objetos

Esta es una diferencia importante y que debemos tener clara. Vamos a ver primero un ejemplo y luego lo explicamos. Imaginemos un objeto coche, un coche tiene una serie de propiedades y de métodos, por ejemplo:

Coche
Matricula
Bastidor
Color
Marca
Modelo
...
conducir(double km) //conduce unos Km
repostar(double litros) //pone gasolina
imprimir() //muestra los datos

En una clase definimos el nombre, las propiedades y los métodos (comportamiento) que van a tener todos los objetos de tipo coche. Podríamos ver una clase como una plantilla de la que se van a crear los objetos. Cuando nosotros creamos un coche concreto (instanciamos), por ejemplo, un Ford Fiesta con matrícula 1234CFV, bastidor: 1243gj4589839, de color rojo; eso es un objeto. Realmente se dice que hemos instanciado un coche creando uno en concreto.

Lo mismo pasaría con la clase Persona, la clase definiría lo que es una persona, pero cuando la instanciamos creando una persona en concreto, distinta de las demás, eso es un objeto.

Veamos un ejemplo simple:

```

3  public class Persona {
4
5      String nombre;
6      String apellidos;
7      int edad;
8      boolean jubilado;
9
10     public void calcularRetencion(){
11         int retencion;
12         if(edad>67 && jubilado)
13             retencion=50;
14         else
15             retencion=80;
16         System.out.println(retencion);
17     }
18
19     public void mostrarDatos(){
20         System.out.println("Nombre: "+nombre+" Apellidos: "+apellidos+" Edad: "+edad+" Jubilado: "+jubilado);
21     }
22
23 }
```

Ilustración 152. Primera aproximación clases

```

2 public class Prueba {
3
4     public static void main (String[] args) {
5
6         Persona pepe=new Persona();
7         pepe.nombre="Pepe";
8         pepe.apellidos="Gotera Callo";
9         pepe.edad=67;
10        pepe.jubilado=true;
11        pepe.mostrarDatos();
12        pepe.calcularRetencion();
13        pepe.edad=68;
14        pepe.calcularRetencion();
15
16    }
17 }
```

Ilustración 153. Test Clases 1

```
C:\Windows\SYSTEM32\cmd.exe
Nombre: Pepe Apellidos: Gotera Callo Edad: 67 Jubilado: true
80
50
```

Ilustración 154. Salida Clases 1

En el primer código podemos ver la declaración de la clase. Defino la clase llamada *Persona* con sus propiedades y dos métodos, uno muestra los datos y otro calcula la retención dependiendo de la edad y si está jubilado. El segundo código es el programa principal (main), podemos observar como en la línea 6 instanciamos la clase persona creando un objeto llamado *pepe*, para ello usamos el operador *new*. Su sintaxis es: <clase> nombreObjeto=new <clase>(). Tras declararlo le damos valores a las propiedades y para ello usamos el “.”, el punto se usa para separar el nombre de la clase del elemento al cual queremos acceder, sea propiedad o método. De esta forma poniendo el nombre del objeto y después un punto podemos ir accediendo a sus propiedades tanto para verlas como para modificarlas y a sus métodos para usarlos. En la salida podemos ver como se muestran los datos del objeto y la retención con 67 años y su modificación tras aumentar los mismos.

Si en ese código creamos otra persona por ejemplo Pedro, las propiedades y métodos de Pepe y Pedro están guardados en posiciones distintas de memoria, el cambiar algo o actuar sobre uno de ellos no repercute sobre el otro, sólo usan la misma clase (plantilla) para crearse.



- Java permite declarar más de una clase en un único fichero, pero ese es tema del siguiente manual, de momento cada clase debe de guardarse en un fichero con “exactamente” el mismo nombre. Si la clase se llama “Pepe” se tiene que guardar en el fichero Pepe.java.
- Es una buena práctica y denota conocimiento, que las clases comiencen con la primera letra en mayúsculas.
- Dado que estamos usando Geany en modo súper simple, para que cuando instanciamos una clase la encuentre la guardaremos en la misma carpeta. En próximos manuales veremos el uso de paquetes con otros IDE.
- Todos los objetos se instancian con el operador new

### 13.2 ¿Qué pasos sigue java para crear un objeto?

A la hora de crear un objeto Java sigue una serie de pasos:

1. Carga el fichero en memoria.
2. Ejecuta los inicializadores *static*.
3. Crea el objeto
  - a. Asigna la memoria necesaria para el objeto mediante el operador *new*.
  - b. Inicializa propiedades.
  - c. Inicializadores del objeto.
  - d. Ejecuta el constructor.

Los inicializadores *static* son un bloque de código que se ejecutará una sola vez al usar la clase, a diferencia del constructor que luego veremos.

Este tipo de inicializadores no devuelven ningún valor, son métodos sin nombre y entre otras cosas se suelen usar para inicializar objetos complejos, inicializan propiedades *static*, y permiten gestionar las excepciones.

### 13.3 Constructores

En la primera aproximación de la clase Persona hemos visto como se crea el objeto y luego le damos valores a sus propiedades, pero sería interesante poder crear un objeto ya con los datos que queremos.

Un constructor sirve para que Java reserve la memoria que necesita para el objeto e inicializar este. Nos podemos encontrar con dos casos:

- **Constructor por defecto (primera aproximación):** No especificamos nosotros el constructor, de manera automática Java ejecuta el constructor por defecto para reservar memoria e inicializar el objeto.
- **Constructor definido:** Nosotros especificamos un constructor en la clase que se ejecutará cada vez que se cree un objeto, Java primero reservará la memoria lo inicializará todo y luego lo ejecutará de forma automática. El constructor debe de tener el **mismo nombre que la clase**, puede recibir parámetros y siempre tiene que ser *public*, hay muchos programadores que omiten el modificador de acceso.

Veamos como cambiaria sabiendo esto nuestro ejemplo:

```

3  public class Persona {
4      String nombre;
5      String apellidos;
6      int edad;
7      boolean jubilado;
8
9      Persona(String nom, String ape, int ed, boolean jub){
10         nombre=nom;
11         apellidos=ape;
12         edad=ed;
13         jubilado=jub;
14     }
15     Persona(String nom, String ape){
16         nombre=nom;
17         apellidos=ape;
18     }
19     Persona(){
20     }
21
22     public void calcularRetencion(){
23         int retencion;
24         if(edad>67 && jubilado)
25             retencion=50;
26         else
27             retencion=80;
28         System.out.println(retencion);
29     }
30     public void mostrarDatos(){
31         System.out.println("Nombre: "+nombre+" Apellidos: "+apellidos+" Edad: "+edad+" Jubilado: "+jubilado);
32     }
33 }
34
35 }
```

Ilustración 155. Segunda aproximación clases

```

2  public class Prueba {
3
4      public static void main (String[] args) {
5
6          Persona pepe=new Persona("Pepe","Gotera Callo",67,true);
7
8          pepe.mostrarDatos();
9          pepe.calcularRetencion();
10         pepe.edad=68;
11         pepe.calcularRetencion();
12
13     }
14 }
15
16 }
```

Ilustración 156. Test Clases 2

```

C:\Windows\SYSTEM32\cmd.exe
Nombre: Pepe Apellidos: Gotera Callo Edad: 67 Jubilado: true
80
50

```

Ilustración 157. Salida clases 2

Como podéis observar el constructor funciona como un método normal en el sentido de que recibe unos parámetros de un tipo y en un orden y lo que hace es asignar esos valores a las propiedades, evitando tener que hacerlo luego. Un constructor también puede realizar operaciones. Esta forma de crear los objetos es la más usada, de hecho, es recomendable poner un constructor vacío si no se necesita inicializar nada, pues en aplicaciones más complejas y de Java EE son necesarios.

En Java pueden existir, en una clase, más de un constructor con el mismo nombre siempre y cuando el tipo y número de parámetros que recibe no sea el mismo, esto es muy útil pues nos

permite crear objetos de distinta forma dependiendo de los datos que usemos como parámetros.

Veamos un ejemplo:

```

3  public class Persona {
4      private String nombre;
5      private String apellidos;
6      private int edad;
7      private boolean jubilado;
8
9      public Persona(String nom, String ape, int ed, boolean jub){
10         nombre=nom;
11         apellidos=ape;
12         edad=ed;
13         jubilado=jub;
14     }
15     public Persona(String nom, String ape){
16         nombre=nom;
17         apellidos=ape;
18     }
19     public Persona(){
20     }
21
22     public int calcularRetencion(){
23         int retencion;
24         if(edad>67 && jubilado)
25             retencion=50;
26         else
27             retencion=80;
28
29         return retencion;
30     }
31
32
33     public void mostrarDatos(){
34         System.out.println("Nombre: "+nombre+" Apellidos: "+apellidos+" Edad: "+edad+" Jubilado: "+jubilado);
35     }
36
37 }
```

Ilustración 158. Tercera aproximación

```

2  public class Prueba {
3
4      public static void main (String[] args) {
5
6          Persona pepe=new Persona("Pepe","Gotera Callo",67,true);
7          Persona luis=new Persona("Pepe","Gotera Callo");
8          Persona pedro=new Persona();
9
10
11      }
12
13 }
```

Ilustración 159. Test Clases 3

Como se puede observar existen tres constructores, el primero necesita todos los datos, el segundo solo el nombre y apellidos, el último nada. De esta forma *pepe* tiene todos sus datos, *luis* solo el nombre y apellidos, no sabemos su edad ni si está jubilado, de *pedro* no sabemos nada.



Algunos lenguajes como C++ tienen destructores para liberar memoria, cerrar conexiones, ... antes de terminar el programa. Java NO tiene destructores, existe un método llamado *finalize* con el fin de liberar memoria llamando al *garbage collector* (recolector de basura), pero en Java no se puede predecir cuándo se va a ejecutar ese proceso, podríamos decir que pasa cuando “le da la gana”, por todo esto lo mejor es dejarlo en manos de la máquina virtual java.

### 13.4 La referencia this

Para hacer referencia al objeto que se está ejecutando usamos *this*. Esta referencia se suele usar cuando puede haber una confusión entre propiedades y métodos, por ejemplo, si vemos el código de la 3<sup>a</sup> aproximación visto antes en el constructor no hay posibilidad de que java confunda los parámetros con las propiedades:

```
 1 public class Persona {
 2     private String nombre;
 3     private String apellidos;
 4     private int edad;
 5     private boolean jubilado;
 6
 7     public Persona(String nom, String ape, int ed, boolean jub){
 8         nombre=nom;
 9         apellidos=ape;
10         edad=ed;
11         jubilado=jub;
12     }
13 }
```

Ilustración 160. Constructor sin this

A los parámetros formales les hemos puesto nombres distintos a las propiedades, pero la mayoría de veces esto no es una buena práctica pues los nombres no son representativos y por tanto no ayudamos a la legibilidad del código. Una mejor opción hubiera sido la siguiente:

```
 3  public class Persona {
 4      private String nombre;
 5      private String apellidos;
 6      private int edad;
 7      private boolean jubilado;
 8
 9      public Persona(String nombre, String apellidos, int edad, boolean jubilado){
10          nombre=nombre;
11          apellidos=apellidos;
12          edad=edad;
13          jubilado=jubilado;
14      }
15  }
```

Ilustración 161. Constructor sin this, mal

Como podéis ver esto es más legible, pero tiene un pequeño problema: cuando hacemos dentro del constructor `nombre=nombre` Java no sabe distinguir a qué nos referimos en cada parte de la asignación, de hecho, saberlo lo sabe, se cree que nos referimos a los parámetros. Para resolver este problema usamos el *this*.

```

3  public class Persona {
4      private String nombre;
5      private String apellidos;
6      private int edad;
7      private boolean jubilado;
8
9      public Persona(String nombre, String apellidos, int edad, boolean jubilado){
10         this.nombre=nombre;
11         this.apellidos=apellidos;
12         this.edad=edad;
13         this.jubilado=jubilado;
14     }

```

Ilustración 162. Constructor con this

Al anteponer al nombre de la propiedad el *this* (separándolo con un punto) le estamos indicando “yo mismo”, en otras palabras, hace referencia al propio objeto que se creará en el *main*, de esta forma siempre que pongamos el *this* delante sabe que nos referimos a las propiedades ya que esos nombres referidos al propio objeto solo pueden ser estas. De esta forma evitamos confusiones y hacemos el código legible.

La referencia a *this* también se puede usar para llamar a un constructor desde otro constructor o método, aunque no suele ser habitual por legibilidad y encapsulación.

*this(parámetros);*



Cuando java busca el nombre de una “variable” lo hace de fuera hacia dentro de los métodos. Primero mira si el nombre es una variable local en el método, si existe entiende que nos referimos a la variable, si no la encuentra busca en los parámetros, si lo encuentra nos estamos refiriendo a ese parámetro, si no lo busca en las propiedades. Si no lo encuentra es cuando ya da error. Con el *this* hacemos que siempre sepa que nos referimos a alguna propiedad.



**Cuidado:** si llamamos a una variable local igual que a un parámetro estaremos ocultando el acceso al parámetro y no podremos usarlo.

### 13.5 Modificadores de acceso

En la programación orientada a objetos las clases nos permiten abstraer el problema a resolver ocultando los datos y como solucionamos el problema. Una clase o un método de una clase no tiene por qué saber cómo se ha solucionado un problema, por ejemplo: durante el manual hemos usado la clase Scanner y sus métodos, pero yo no sé cómo están hechos. Otra cosa muy importante de la POO es que no se debe de poder acceder directamente a las propiedades de un objeto desde otro. Por todo esto existen unos modificadores de acceso que nos indican quien puede acceder a que método o propiedad:

- **Público (public):** Un método o propiedad pública puede ser accedida desde cualquier otra clase o subclase que se necesite.

- **Privado (private):** Solo se puede acceder desde dentro de la propia clase.
- **Protegido (protected):** Se pueden acceder desde clases o subclases que están en el mismo paquete o carpeta.

Modificador de acceso	public	protected	private	Sin especificar
¿El método o propiedad es accesible desde su propia clase?	SÍ	SÍ	SÍ	SÍ
¿El método o propiedad es accesible desde otras clases en el mismo paquete (carpeta)?	SÍ	SÍ	NO	SÍ
¿El método o propiedad es accesible desde una subclase en el mismo paquete (carpeta)?	SÍ	SÍ	NO	SÍ
¿El método o propiedad es accesible desde subclases en otros paquetes?	SÍ	*	NO	NO
¿El método o propiedad es accesible desde otras clases en otros paquetes?	SÍ	NO	NO	NO

(\*) No se suele dar. Se podría acceder a los miembros desde objetos de las subclases, pero no de la superclase (veremos qué es esto más adelante)

Tabla 25. Modificadores de acceso

Veamos cómo aplicar esto a nuestro ejemplo:

```

3  public class Persona {
4      private String nombre;
5      private String apellidos;
6      private int edad;
7      private boolean jubilado;
8
9      public Persona(String nom, String ape, int ed, boolean jub){
10         nombre=nom;
11         apellidos=ape;
12         edad=ed;
13         jubilado=jub;
14     }
15     public Persona(String nom, String ape){
16         nombre=nom;
17         apellidos=ape;
18     }
19     public Persona(){
20     }
21
22     private int calcularRetencion(){
23         int retencion;
24         if(edad>67 && jubilado)
25             retencion=50;
26         else
27             retencion=80;
28
29         return retencion;
30     }
31
32     public void mostrarRetencion(){
33         int retencion;
34         retencion=calcularRetencion();
35         System.out.println(retencion);
36     }
37
38
39     public void mostrarDatos(){
40         System.out.println("Nombre: "+nombre+" Apellidos: "+apellidos+" Edad: "+edad+" Jubilado: "+jubilado);
41     }
42 }
43

```

Ilustración 163. Cuarta aproximación

```

2  public class Prueba {
3
4      public static void main (String[] args) {
5
6          Persona pepe=new Persona("Pepe","Gotera Callo",67,true);
7          Persona luis=new Persona("Pepe","Gotera Callo");
8          Persona pedro=new Persona();
9
10         pedro.nombre="Pedro";
11     }
12 }
13

```

Estado: javac "Prueba.java" (en el directorio: D:\DAM1\programacion\varios\codigo manual\Clases\Ej3)  
 Compilador: Prueba.java:10: error: nombre has private access in Persona  
 pedro.nombre="Pedro";  
 ^  
 1 error  
 Ha fallado la compilación.

Ilustración 164. Acceso erróneo

Como podemos ver en la clase Persona hemos añadido los modificadores de acceso. Las propiedades son de tipo *private*, el modificador se pone a la izquierda del tipo. Al ser *private* no puede ser accedido por ninguna otra clase. Los constructores son *public* de forma que pueden ser accedidos desde cualquier clase, pensad que es lógico ya que se tienen que poder ejecutar al construir un objeto. (*en este caso no es necesario ponerlo ya que al no poner nada actúan de igual forma. Es importante recalcar que, aunque es una práctica común no ponerlo en los constructores, en los demás métodos es conveniente especificar siempre el modificador de manera explícita.*)

Para ver un ejemplo de un método he dividido el cálculo de la retención en dos métodos, uno realiza el cálculo y el otro devuelve el valor. Este método es llamado desde *mostrarRetencion()*. Este último es *public* ya que tiene que poder ser llamado desde otras clases, pero *calcularRetencion()* solo va a ser llamado por *mostrarRetencion()* por lo que es *private*, nadie desde fuera de la clase tiene acceso a él.

Veamos ahora a la clase Prueba: las líneas de la 6 a la 8 se ejecutarán sin problemas, pero la línea 10 está intentando acceder a una propiedad que es privada, el compilador se da cuenta y nos avisa de que no puede hacerse, no dejando compilar la clase. Esto sucedería tanto para guardar algo en una propiedad, como es el caso, como para leerla.

Por norma no se debe de tener acceso a la estructura interna de las clases, esto implica que las propiedades serán privadas y aquellos métodos que no sea necesario usar desde fuera de la clase también. Pero, si las propiedades de un objeto tienen que ser privadas ¿Cómo puedo modificarlas o consultarlas?, muy simple, con los llamados *getter's* y *setter's*. Un *getter* es un método, público que me devuelve el valor de una propiedad y un *setter* es un método, también público, que me permite cambiar el valor de una propiedad. El nombre de estos métodos se forma de la siguiente forma:

El prefijo *set* o *get* seguido del nombre de la propiedad teniendo en cuenta que el primer carácter de esta se escribe en mayúsculas. Nosotros podríamos crear los métodos con los nombres que quisiéramos, pero si hacemos eso el código será menos legible y opciones más

avanzadas de Java SE o Java EE no funcionarán correctamente. Por tanto, sigamos las buenas prácticas de Java, veamos un ejemplo:

```

2  public class Persona {
3
4      private String nombre;
5      private String apellidos;
6      private int edad;
7
8      public Persona(String nombre, String apellidos, int edad) {
9          this.nombre = nombre;
10         this.apellidos = apellidos;
11         this.edad = edad;
12     }
13
14     public String getNombre() {
15         return nombre;
16     }
17
18     public void setNombre(String nombre) {
19         this.nombre = nombre;
20     }
21
22     public String getApellidos() {
23         return apellidos;
24     }
25
26     public void setApellidos(String apellidos) {
27         this.apellidos = apellidos;
28     }
29
30     public int getEdad() {
31         return edad;
32     }
33
34     public void setEdad(int edad) {
35         this.edad = edad;
36     }
37 }
```

Ilustración 165. Setter's y getter's

Para solucionar el error de compilación del ejemplo anterior usaríamos:

```
pedro.setNombre("Pedro");
```

### 13.6 Static

En Java no existen variables globales, esto significa que para que todos los objetos de una clase compartan una única propiedad debe de ser declarada como static.

```

3  public class Pajaro
4  {
5
6      public static int numPajaros;
7      private String color;
8      private String nombre;
9      private double peso;
10     private double altura;
11
12     pajaro(String color, String nombre, double peso, double altura)
13     {
14         this.color = color;
15         this.nombre = nombre;
16         this.peso = peso;
17         this.altura = altura;
18         numPajaros++;
19     }
20 }
```

En el ejemplo anterior vemos como la propiedad *numPajaros* está declarada como *static* (se pone después del modificador de acceso), esto hace que todos los objetos de tipo *Pajaro*

compartan la misma propiedad, en la misma posición de memoria, lo que significa que si un objeto pájaro la modifica lo hace para todos los pájaros. En el ejemplo podemos ver como el constructor tiene un incremento de esa propiedad, `numPajaros++`, esto hace que cada vez que se ejecute, se cree un pájaro e incremente el número en 1, por tanto, por el simple hecho de crear un pájaro se incrementa el número de pájaros. Consultando esa propiedad en cualquier pájaro obtendremos el número de pájaros que tenemos.

El siguiente código crea tres pájaros y consulta el número de pájaros usando uno de ellos, el resultado será 3:

```

3  public class PajaroTest {
4
5      public static void main (String args[]) {
6
7          Pajaro uno=new Pajaro("rojo", "Pepe",3,5);
8          Pajaro dos=new Pajaro("verde", "Pedro",1,2);
9          Pajaro tres=new Pajaro("amarillo", "Eva",2,3);
10         System.out.println(dos.getNumPajaros());
11     }
12 }
13

```

Ilustración 166. Propiedad static

Da igual quien llame al método `getNumPajaros`, el resultado será el mismo.

Los métodos con `static` son llamados de clase y los métodos “normales” se llaman de instancia.



#### Reglas:

- Los métodos `static` no tienen referencia `this`.
- Un método `static` no puede acceder a miembros que no sean `static`
- Un método no `static` puede acceder a miembros `static` y no `static`



La clase `Math` del paquete `java.lang` donde se encuentran los métodos para realizar operaciones matemáticas, se conoce como una clase estática, pues todos sus métodos son `static`. Para calcular una raíz cuadrada no instanciamos un objeto simplemente usamos el nombre de la clase y el método a usar: `Math.sqrt(25)`. Veremos esto en los anexos.

## 13.7 Final

Java no tiene constantes como tal, pero tenemos una forma de hacer que una propiedad de un objeto no pueda cambiar. Para ello usamos el modificador final.

```

...
public static final int PATAS=2;
private String color;
private String nombre;
private double peso;
private double altura;
...

```

Ilustración 167. Modificador final

Cuando usamos final después del modificador de acceso le estamos diciendo que esa propiedad no puede ser cambiada. Cuando se declara no es necesario darle un valor siempre pero sólo se puede hacer una única vez. Lo habitual es inicializarla en la misma declaración. PATAS vale 2 para siempre. Es habitual usar *static* junto con *final* para que todos los objetos comparten la misma dirección de memoria de esa propiedad ya que esta no va a cambiar. Es habitual y de buena praxis escribir las propiedades *final* en mayúsculas.

### 13.8 Vectores de objetos

Obviamente se pueden crear vectores de objetos, por ejemplo, un vector de coches. Los vectores de objetos se inicializan de forma predefinida a null. Null significa vacío, no existe objeto y es una palabra reservada de Java que se muestra de color azul en Geany. Debemos tener cuidado ya que si intentamos hacer algo con una casilla que está a null nos dará un error de ejecución ya que, por ejemplo, no puede llamar a un método de un objeto que no existe.

En el siguiente ejemplo se crea un vector de coches dado su modelo y los kilómetros que tiene, luego se conduce 10 kilómetros en todos. Veamos lo que sucede:

```

3  public class Coche {
4
5      private String modelo;
6      private int kilometros;
7
8      public Coche(String modelo, int kilometros) {
9          this.modelo = modelo;
10         this.kilometros = kilometros;
11     }
12
13     public String getModelo() {
14         return modelo;
15     }
16
17     public int getKilometros() {
18         return kilometros;
19     }
20
21     public void conducir(int km){
22         kilometros=kilometros+km;
23     }
24 }

```

Ilustración 168. Clase coche

```

3  public class Test {
4
5
6      public static void main(String[] args) {
7
8          Coche [] c=new Coche[4];
9
10         c[0]=new Coche("Fiesta",10);
11         c[1]=new Coche("Ibiza",30);
12         c[3]=new Coche("Cupra",0);
13
14         for (int i = 0; i < c.length; i++) {
15             c[i].conducir(10);
16         }
17     }
18 }
19
20 }
```

Ilustración 169. main

C:\Windows\SYSTEM32\cmd.exe  
Exception in thread "main" java.lang.NullPointerException  
at Test.main(Test.java:15)  
-----  
(program exited with code: 1)  
Presione una tecla para continuar . . .

Ilustración 170. Salida null

Como podemos ver se crea un vector de 4 pero me dejo por crear el coche en la posición 2, esa casilla está a null. Cuando el for las recorre y llega a la 2 intenta hacer null.conducir(10) y eso causa una excepción ya que no hay coche (se encuentra con un null).

## 14. Herencia

La Herencia es uno de los 4 pilares de la programación orientada a objetos (POO) junto con la Abstracción, Encapsulación y Polimorfismo. La herencia es un mecanismo que permite la definición de una clase a partir de la definición de otra ya existente. La herencia permite compartir automáticamente métodos y datos entre clases, subclases y objetos. Otra forma de ver la herencia es la de “sacar factor común” al código que escribimos.

El ejemplo que proponemos es un caso en el que vamos a simular el comportamiento que tendrían los diferentes integrantes de la selección española de fútbol; tanto los Futbolistas como el cuerpo técnico (Entrenadores, Masajistas, etc...). Para simular este comportamiento vamos a definir tres clases que van a representar a los objetos Futbolista, Entrenador y Masajista. De cada uno de ellos vamos a necesitar algunos datos que reflejaremos en las propiedades y una serie de acciones que reflejaremos en sus métodos. Estos atributos y métodos los mostramos en el siguiente diagrama de clases:

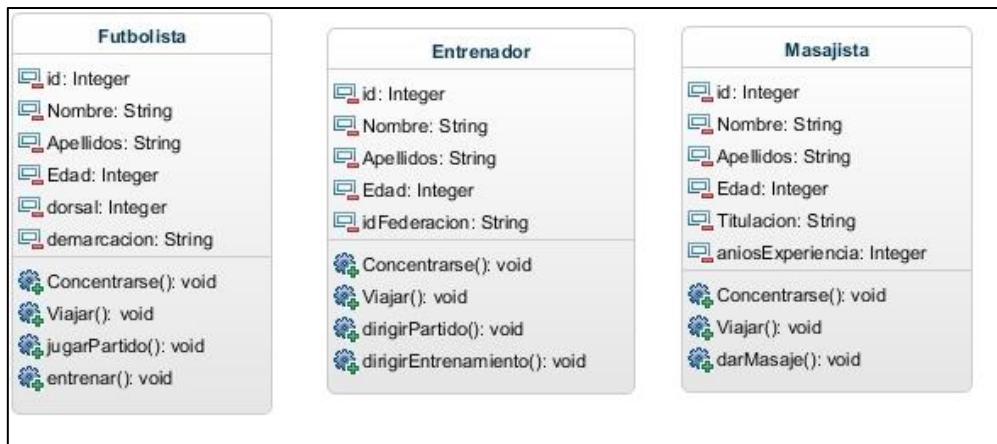


Ilustración 171. Diagrama herencia

Como se puede observar, vemos que en las tres clases tenemos propiedades y métodos que son iguales:

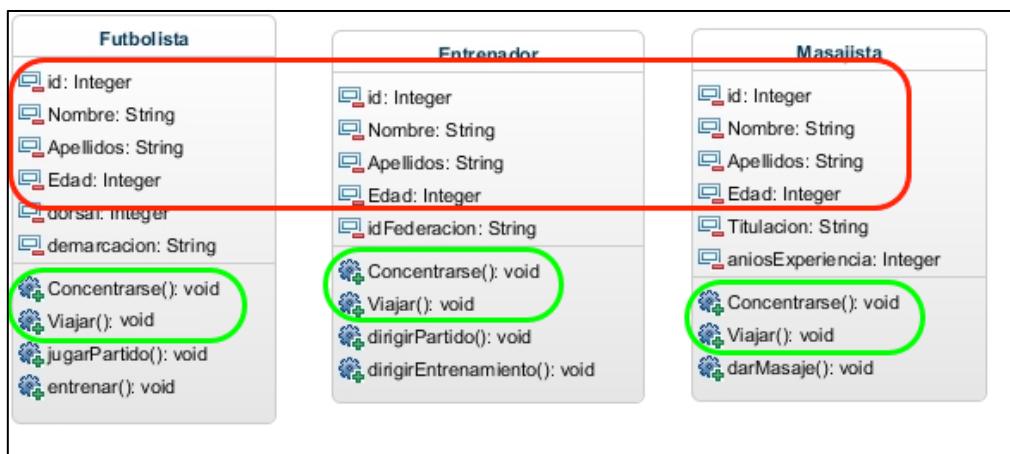


Ilustración 172. Factor común en herencia

Lo que podemos ver en este punto es que estamos escribiendo mucho código repetido ya que las tres clases tienen métodos y propiedades comunes, de ahí que decimos "sacar factor común" para no escribir código repetido, por tanto lo que haremos será crearnos una clase con el "código que es común a las tres clases" (a esta clase se le denomina en la herencia como "Clase Padre o SuperClase") y el código que es específico de cada clase, lo dejaremos en ella, siendo denominadas estas clases como "Clases Hijas", las cuales heredan de la clase padre sus propiedades y métodos.

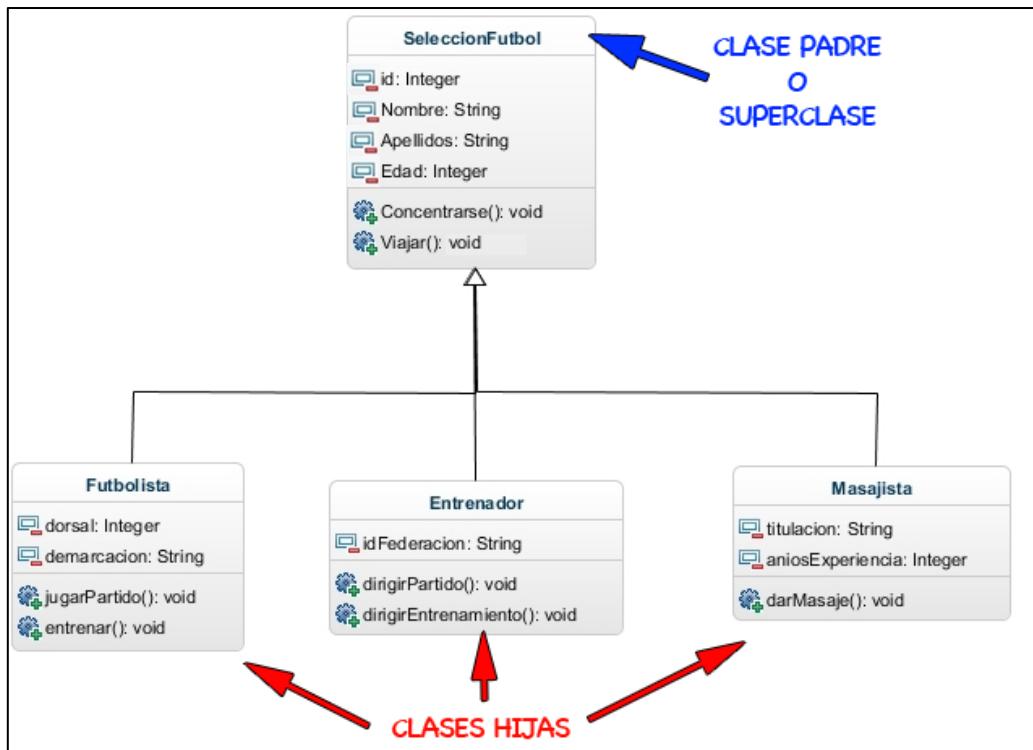


Ilustración 173. Clases padre e hijas

Como podéis observar ahora queda un código mucho más limpio, estructurado, con menos líneas de código, lo que lo hace más legible y reutilizable.

Veamos cómo quedaría el código:

En la primera clase, la padre, podemos ver que no hay nada nuevo simplemente tiene la sintaxis de una clase normal y corriente. Para facilitar el código y centrarnos en lo que nos ocupa he imprimido por pantalla desde dentro de las clases pero como ya sabéis esto **no debe de hacerse y es una muy mala praxis** que va en contra del encapsulamiento y la reutilización. **Lo usamos sólo a modo de laboratorio.**

En esta clase hemos puesto las propiedades y métodos que son comunes a todas. Tanto un Futbolista como un Entrenador o Masajista deben de poder viajar y concentrarse. De todos ellos queremos saber su id, nombre, apellidos y edad.

```
1  public class SeleccionFutbol {  
2      private int id;  
3      private String nombre;  
4      private String apellidos;  
5      private int edad;  
6  
7      public SeleccionFutbol(int id, String nombre, String apellidos, int edad) {  
8          this.id = id;  
9          this.nombre = nombre;  
10         this.apellidos = apellidos;  
11         this.edad = edad;  
12     }  
13  
14     public int getId() {  
15         return id;  
16     }  
17  
18     public void setId(int id) {  
19         this.id = id;  
20     }  
21  
22     public String getNombre() {  
23         return nombre;  
24     }  
25  
26     public void setNombre(String nombre) {  
27         this.nombre = nombre;  
28     }  
29  
30     public String getApellidos() {  
31         return apellidos;  
32     }  
33  
34     public void setApellidos(String apellidos) {  
35         this.apellidos = apellidos;  
36     }  
37  
38     public int getEdad() {  
39         return edad;  
40     }  
41  
42     public void setEdad(int edad) {  
43         this.edad = edad;  
44     }  
45  
46     public void concentrarse() {  
47         System.out.println("Concentrarse");  
48     }  
49  
50     public void viajar() {  
51         System.out.println("Viajar");  
52     }  
53 }
```

Ilustración 174. Ejemplo clase padre

```

1  public class Masajista extends SeleccionFutbol {
2
3     private String titulacion;
4     private int aniosExperiencia;
5
6     public Masajista(int id, String nombre, String apellidos, int edad, String titulacion, int aniosExperiencia) {
7         super(id, nombre, apellidos, edad);
8         this.titulacion = titulacion;
9         this.aniosExperiencia = aniosExperiencia;
10    }
11
12    public String getTitulacion() {
13        return titulacion;
14    }
15
16    public void setTitulacion(String titulacion) {
17        this.titulacion = titulacion;
18    }
19
20    public int getAniosExperiencia() {
21        return aniosExperiencia;
22    }
23
24    public void setAniosExperiencia(int aniosExperiencia) {
25        this.aniosExperiencia = aniosExperiencia;
26    }
27
28    public void darMasaje() {
29        System.out.println("Da un masaje");
30    }
31
32 }

```

Ilustración 175. Clase hija 1

En la clase *Masajista* ya podemos ver cambios:

- Usamos la palabra reservada *extends* para indicar que hereda de *SeleccionFutbol*.
- Sólo declara las propiedades y métodos que necesita por ser *Masajista*, las demás ya las ha declarado el padre.
- El constructor recibe todas las propiedades que necesita sólo que unas las tiene *Masajista* y otras su padre. Para indicar que las que no son del hijo se deben de enviar al padre se usa la palabra reservada *super* y entre paréntesis las propiedades de las que se encarga el padre. Esto ejecuta el constructor del padre con esas propiedades. Fíjate como coinciden el número de parámetros y tipo con el constructor del padre. El *super* debe de ser la primera línea del constructor.

Las demás clases hacen exactamente lo mismo. Envían al padre lo que es del padre y se quedan con lo que es suyo.

```

1  public class Futbolista extends SeleccionFutbol {
2
3      private int dorsal;
4      private String demarcacion;
5
6      public Futbolista(int id, String nombre, String apellidos, int edad, int dorsal, String demarcacion) {
7          super(id, nombre, apellidos, edad);
8          this.dorsal = dorsal;
9          this.demarcacion = demarcacion;
10     }
11
12     public int getDorsal() {
13         return dorsal;
14     }
15
16     public void setDorsal(int dorsal) {
17         this.dorsal = dorsal;
18     }
19
20     public String getDemarcacion() {
21         return demarcacion;
22     }
23
24     public void setDemarcacion(String demarcacion) {
25         this.demarcacion = demarcacion;
26     }
27
28     public void jugarPartido() {
29         System.out.println("Juega un partido");
30     }
31
32     public void entrenar() {
33         System.out.println("Entrena");
34     }
35
36 }

```

Ilustración 176. Clase hija 2

```

2  public class Entrenador extends SeleccionFutbol {
3
4      private String idFederacion;
5
6      public Entrenador(int id, String nombre, String apellidos, int edad, String idFederacion) {
7          super(id, nombre, apellidos, edad);
8          this.idFederacion = idFederacion;
9      }
10
11     public String getIdFederacion() {
12         return idFederacion;
13     }
14
15     public void setIdFederacion(String idFederacion) {
16         this.idFederacion = idFederacion;
17     }
18
19     public void dirigirPartido() {
20         System.out.println("Dirige un partido");
21     }
22
23     public void dirigirEntrenamiento() {
24         System.out.println("Dirige un entrenamiento");
25     }
26
27 }

```

Ilustración 177. Clase hija 3

```

2  public class Main {
3
4      public static void main(String[] args) {
5
6          Entrenador delBosque = new Entrenador(1, "Vicente", "Del Bosque", 60, "284EZ89");
7          Futbolista iniesta = new Futbolista(2, "Andrés", "Iniesta", 29, 6, "Interior Derecho");
8          Masajista raulMartinez = new Masajista(3, "Raúl", "Martínez", 41, "Licenciado en Fisioterapia", 18);
9
10         System.out.println(delBosque.getNombre());
11         iniesta.concentrarse();
12         iniesta.entrenar();
13
14     }
15
16 }
17

```

Ilustración 178. Ejemplo de main con herencia

En el *main* podemos ver como se crea un objeto de cada hijo. En la línea 10 imprimimos el nombre del entrenador, en la 11 llamamos a que un jugador se concentre (dicho método no está declarado en el jugador sino en su padre), en la línea 12 llamamos a entrenar.

```
C:\Windows\system32\cmd.exe
Vicente
Concentrarse
Entrena
Presione una tecla para continuar . . .
```

¿Cómo encuentra Java los métodos donde corresponde? ¿Cómo podemos hacer que actúen de forma distinta?

Cuando en el *main* se llama a un método Java lo busca en la clase que estamos buscando, si está lo llama y si no lo busca en el padre, abuelo... hasta llegar a *Object* si entonces no lo encuentra es cuando da error.

Ejemplo:

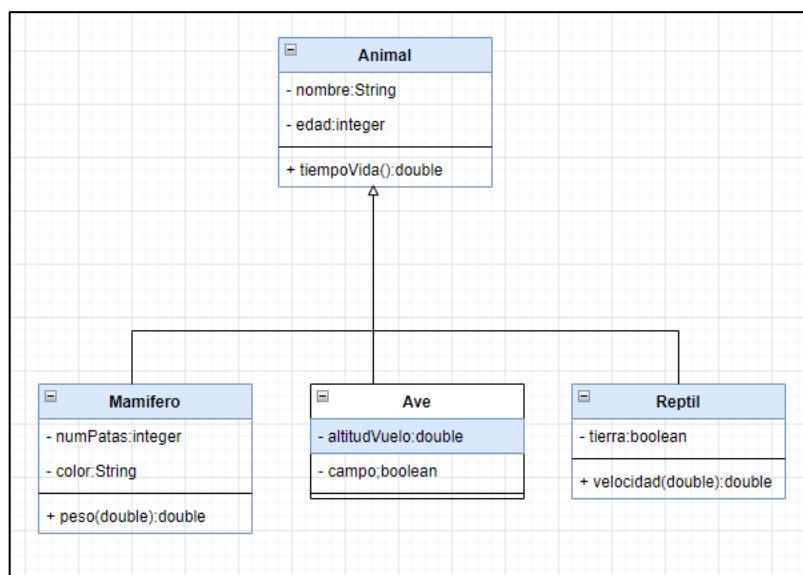


Ilustración 179. Diagrama animales

En este ejemplo podemos ver como tenemos tres tipos de animales que tienen en común que son animales y por tanto tienen un nombre, una edad y un tiempo de vida. Cuando en el *main* cree un *Mamifero* y llame a *tiempoVida()* Java no lo encontrará, por lo que buscará en su padre *Animal*, lo encuentra y lo ejecuta.

#### 14.1 Sobreescritura

Imaginemos que las aves tienen un tiempo de vida que se calcula de forma distinta al resto de animales. Podemos sobreescribir el método en animales con su código:

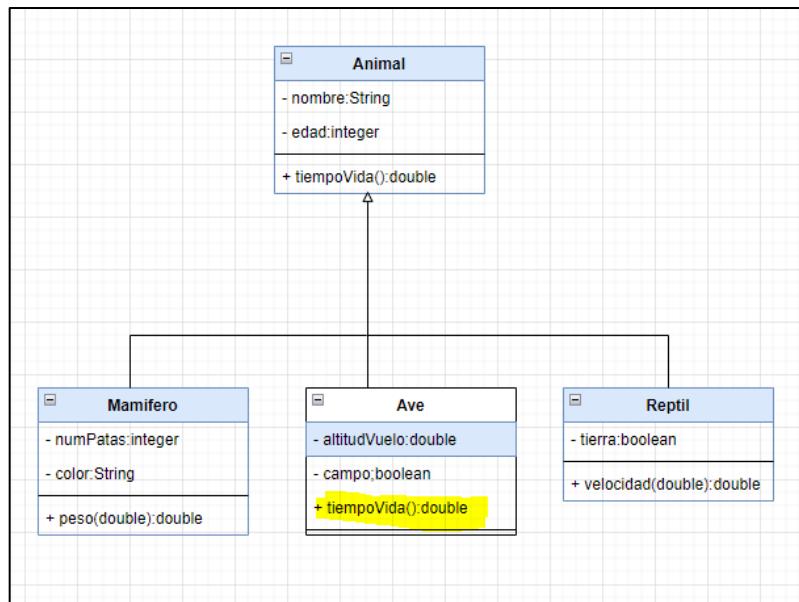


Ilustración 180. Sobreescritura

Hemos añadido un método en **Ave**. Cuando Java lo llame desde **Ave** lo encontrará y no recurrirá al padre.

Para poder sobreescribir un método:

- Tiene que tener el mismo nombre.
- El return debe de ser del mismo tipo.
- Debe de conservar los mismos parámetros.

## 14.2 Sobrecarga

La sobrecarga es cuando podemos tener más de un método con el mismo nombre, como vimos en los constructores.

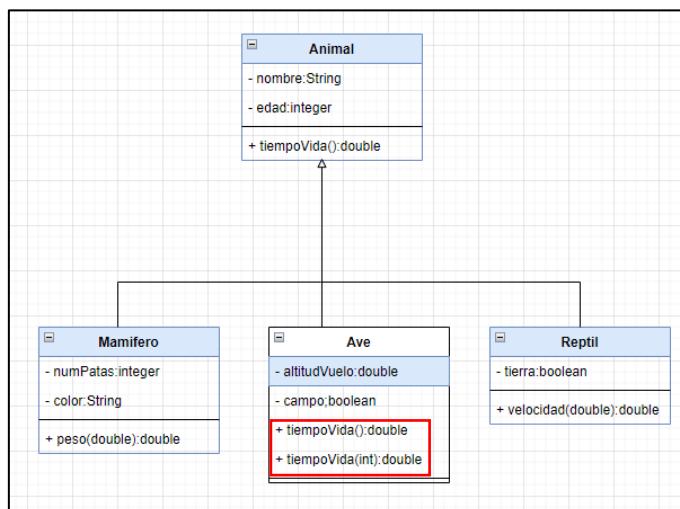


Ilustración 181. Sobrecarga

Dependiendo de si le paso un *int* como parámetro se ejecutará uno u otro. No tienen por qué estar en la misma clase:

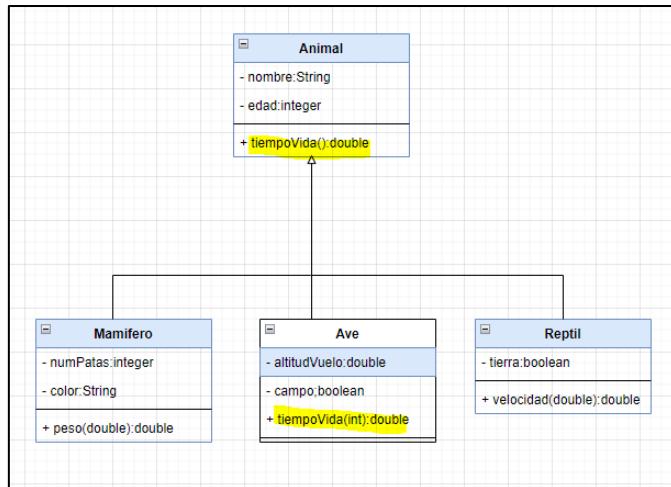


Ilustración 182. Sobrecarga 2

Reglas:

- Los métodos sobrecargados deben de cambiar la lista de argumentos obligatoriamente. No pueden coincidir en número y tipo.
- Se pueden sobrecargar en una misma clase o en subclases.
- Pueden cambiar el tipo de retorno o el modificador de acceso.

### 14.3 `toString`

El `toString` es un método especial que devuelve una representación como `String` del objeto. Se suele usar para mostrar sus propiedades o como queremos representarlo como texto. Lo veremos en más profundidad en el siguiente capítulo.

Pero, ¿Tiene sentido declarar un objeto del padre?

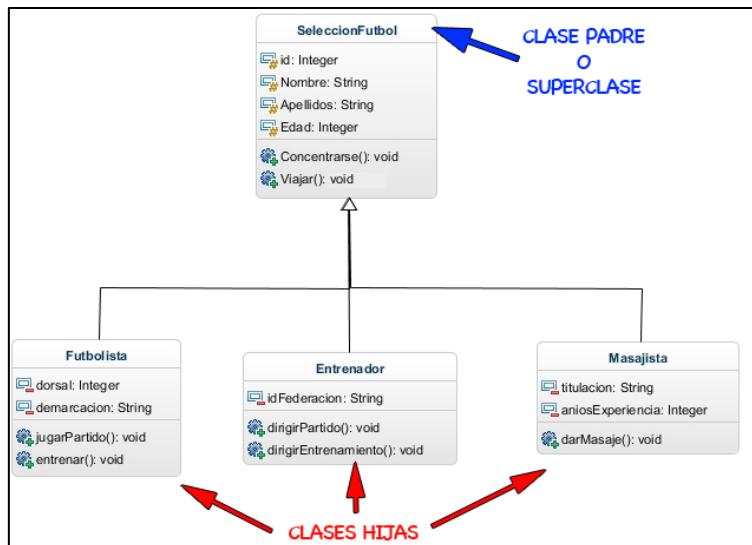
### 14.4 Clases abstractas

Las clases abstractas son aquellas que por sí mismas no se pueden identificar con algo “concreto” (no existen como tal en el mundo real), pero sí poseen determinadas características que son comunes en otras clases que pueden ser creadas a partir de ellas.

Las clases abstractas tienen dos grandes características:

- No se pueden instanciar.
- Pueden contener métodos abstractos. Un método abstracto define la cabecera de un método y que debe de hacer, pero no como. Toda clase que hereda de una clase abstracta está obligada a dar solución a esos métodos.

Veamos el ejemplo anterior del equipo de fútbol:



¿Tiene sentido crear objetos de la clase *SeleccionFutbol*? Nuestro programa va a trabajar con futbolistas, entrenadores y masajistas, por tanto, no tiene sentido crear un objeto del padre. Para encapsular el código y que el programador que realiza el programa principal no cree un objeto de ese tipo, bien por error o por desconocimiento, declaramos la clase padre como abstracta, de esta forma le dará un error de compilación si trata de instanciarla. Para declarar una clase como abstracta se usa la palabra reservada *abstract* antes de *class*.

Por otro lado imaginemos lo siguiente: Todos los hijos deben de declarar un método *nivelCansancio()* que dado las horas trabajadas me devuelva un entero que indique su nivel de cansancio. Las clases abstractas también pueden declarar métodos abstractos, estos métodos no se solucionan en ella, sino que obligan a los hijos a darle solución. Veamos el ejemplo descrito:

```

2 public abstract class SeleccionFutbol {
3
4     private int id;
5     private String nombre;
6     private String apellidos;
7     private int edad;
8
9     public SeleccionFutbol(int id, String nombre, String apellidos, int edad) {
10        this.id = id;
11        this.nombre = nombre;
12        this.apellidos = apellidos;
13        this.edad = edad;
14    }
15
16    public abstract int nivelCansancio(int horas);
17}

```

Ilustración 183. Clase abstracta

Como se puede observar la clase se declara como *abstract*, sólo por eso ya no podemos crear objetos de esta clase, no es instanciable. Independientemente de ello una clase abstracta puede declarar métodos como abstractos. La sintaxis de los métodos abstractos sigue las siguientes reglas:

- No tiene cuerpo (llaves): sólo consta de firma con paréntesis. Entre paréntesis pueden tener parámetros como un método normal.
- Terminan con un punto y coma.

- Sólo puede existir dentro de una clase abstracta. Si una clase incluye un método abstracto, forzosamente la clase será una clase abstracta.
- Los métodos abstractos forzosamente habrán de estar sobreescritos en las clases hijas. Si una subclase no implementa un método abstracto de la superclase a la fuerza tiene que ser abstracta también.
- Las clases que lo solucionan usan la sintaxis normal de los métodos siendo la cabecera la misma.

Veamos cómo se implementaría la clase futbolista, por ejemplo:

```

2  public class Futbolista extends SeleccionFutbol {
3
4      private int dorsal;
5      private String demarcacion;
6
7      public Futbolista(int id, String nombre, String apellidos, int edad, int dorsal, String demarcacion) {
8          super(id, nombre, apellidos, edad);
9          this.dorsal = dorsal;
10         this.demarcacion = demarcacion;
11     }
12
13
14     public int nivelCansancio(int horas){
15         return (horas*25)/2;
16     }
17 }
```

Ilustración 184. Clase abstracta 2

*Obviamente todos los métodos que había en este ejemplo siguen estando, no los he puesto por acortar las imágenes.*

Una clase abstracta puede tener métodos no abstractos.



Si un parente es abstracto y hay clases intermedias abstractas y clases finales que solucionan los métodos abstractos, las intermedias no es necesario que declaren los métodos abstractos.

## 14.5 Polimorfismo

En programación orientada a objetos, polimorfismo es la capacidad que tienen los objetos de una clase en ofrecer respuesta distinta e independiente en función de los parámetros (diferentes implementaciones) utilizados durante su invocación. Dicho de otro modo, el objeto como entidad puede contener valores de diferentes tipos durante la ejecución del programa.

Como siempre veamos un ejemplo que se entenderá mejor. Imaginemos que tenemos productos, por ejemplo trigo y leche. Ambos tienen cosas en común y tienen que solucionar un método para calcular su precio. En el trigo a ese precio se le suma un tanto por cien, en la leche dependiendo del tipo una cantidad fija. Veamos el código y luego lo explico.

```

2  public abstract class Producto {
3
4      private String nombre;
5      private int precio;
6
7      Producto(String nombre, int precio){
8          this.nombre=nombre;
9          this.precio=precio;
10     }
11     public abstract int calularTotal();
12
13     public int getPrecio(){
14         return precio;
15     }
16     public String toString(){
17         return "Nombre:"+nombre+" Precio:"+precio;
18     }
19 }
```

Ilustración 185. Producto

```

3  public class Trigo extends Producto {
4
5      private int ganancia; //en %
6
7      Trigo(String nombre, int precio,int ganancia){
8          super(nombre,precio);
9          this.ganancia=ganancia;
10     }
11
12     public int calularTotal(){
13         return getPrecio()+(getPrecio()*ganancia/100);
14     }
15     public String toString(){
16         return super.toString()+" Ganancia:"+ganancia;
17     }
18 }
19 }
```

Ilustración 186. Trigo

```

1  public class Leche extends Producto {
2
3      private int tipo; //0->normal, 1->semi, 2->desnatada
4
5      Leche(String nombre, int precio,int tipo){
6          super(nombre,precio);
7          this.tipo=tipo;
8      }
9
10     public int calularTotal(){
11         int total=super.getPrecio();
12         switch (tipo){
13             case 0:total+=1;
14                 break;
15             case 1:total+=2;
16                 break;
17             case 2:total+=3;
18                 break;
19         }
20         return total;
21     }
22     public String toString(){
23         return super.toString()+" Tipo:"+tipo;
24     }
25 }
26 }
```

Ilustración 187. Leche

```

2  public class TestPol {
3
4      public static void main (String args[]) {
5
6          Producto p1=new Trigo("Trigo",20,5);
7          Producto p2=new Leche("Leche",10,2);
8
9          System.out.println(p1);
10         System.out.println(p1.calcularTotal());
11         System.out.println(p2);
12         System.out.println(p2.calcularTotal());
13
14     }
15 }
```

Ilustración 188. Test Polimorfismo

- **Producto:** Las propiedades de esta clase son las comunes a todos los productos. Tienen un método abstracto para calcular el precio que deben de solucionar sus hijas. También cuenta con un *toString()* para devolver una representación como texto del objeto que se instancie.
- **Trigo:** Es una clase que hereda de Producto. Añade el tanto por cien que hay que sumarle al precio. Soluciona el método de calcular el precio y también tiene un *toString*. Fíjate como el *toString* llama al del padre primero para poder mostrar todos los datos.
- **Leche:** Hereda de Producto, calcula el precio dependiendo del tipo de leche y tiene un *toString*.
- **TestPol:** En esta clase se crean dos objetos, uno de trigo y otro de leche pero el tipo de objeto es el del parente, *Producto*. *Un parente quiere a todos sus hijos por igual*, en una variable de tipo *Producto* caben productos de tipo *Trigo* y *Leche*. Al revés no se puede hacer pues perderíamos precisión. Dado que ambos tienen el *toString* y calcular precio podemos llamarlos.

```
C:\Windows\system32\cmd.exe
Nombre:Trigo Precio:20 Ganancia:5
21
Nombre:Leche Precio:10 Tipo:2
13
Presione una tecla para continuar . . .
```

Ilustración 189. Salida Polimorfismo

Para una única variable no tiene mucha utilidad, pero ¿y si nos piden guardar 50 productos distintos en un vector? Puedo declarar el vector de tipo *Producto* e ir guardando productos ya que todos caben.

Veamos un ejemplo de un vector pero un poco más complicado. Añadimos un método a cada producto:

```

3  public class Trigo extends Producto {
4
5      private int ganancia; //en %
6
7      Trigo(String nombre, int precio,int ganancia){
8          super(nombre,precio);
9          this.ganancia=ganancia;
10     }
11     public boolean excesivaGanancia(){
12         boolean temp=false;
13         if(ganancia>40)
14             temp=true;
15         return temp;
16     }
17
18     public int calularTotal(){
19         return getPrecio()+(getPrecio()*ganancia/100);
20     }
21     public String toString(){
22         return super.toString()+" Ganancia:"+ganancia;
23     }
24 }
25

```

Ilustración 190. Trigo 2

He añadido a Trigo un método que me dice si la ganancia es excesiva. Para realizarlo he usado una forma muy común en programación, supongo una cosa y demuestro si es verdad. En este caso supongo que es falso y si entra en el *if* lo cambia.

```

10    public String tipoLeche(){
11        String temp="";
12        switch (tipo){
13            case 0:temp="Normal";
14                break;
15            case 1:temp="Semi";
16                break;
17            case 2:temp="Desnatada";
18                break;
19        }
20        return temp;
21    }

```

Ilustración 191. Leche 2

A Leche le he añadido un método que me devuelve el nombre del tipo de leche.

Escribamos el test para devolver el nombre del tipo de leche:

```

2  public class TestPoli {
3
4      public static void main (String args[]) {
5
6          Producto p1=new Trigo("Trigo",20,5);
7          Producto p2=new Leche("Leche",10,2);
8
9          System.out.println(p2.tipoLeche());
10
11     }
12 }

```

Ilustración 192. Error Polimorfismo

```

Estado z:@JoseA\GeanyPortable\jdk8\bin\javac "TestPoli.java"
Compilador TestPoli.java:9: error: cannot find symbol
Mensajes System.out.println(p2.tipoLeche());
Borrador ^
symbol: method tipoLeche()
location: variable p2 of type Producto
1 error

```

Ilustración 193 Error Polimorfismo 2

¿Qué está sucediendo? Pese a que en la variable hay un producto de tipo *Leche* el tipo de la variable es *Producto* y la clase *Producto* no tiene ningún método *tipoLeche*.

Esto tiene dos soluciones:

1. Crear el método, aunque sea abstracto en la clase *Producto*. Esto no es buena opción ya que si tenemos muchos tipos de productos deberíamos añadir todos los métodos en el padre cuando solo lo usa un hijo.
2. Usar la palabra reservada *instanceof*: *instanceof* me dice si un objeto pertenece a una clase (se puede hacer de otras formas pero está es la más fácil bajo mi punto de vista). Una vez sabemos que sí pertenece a una clase hacemos un casting de los vistos el principio del manual y ya podemos usar el método pues tendremos un objeto tipo *Leche*. Veamos un ejemplo:

```

2 public class TestPoli {
3
4     public static void main (String args[]) {
5
6         Producto p1=new Trigo("Trigo",20,5);
7         Producto p2=new Leche("Leche",10,2);
8
9         if(p2 instanceof Leche){
10             Leche temp=(Leche)p2;
11             System.out.println(temp);
12         }
13     }
14 }
```

Ilustración 194. *instanceof*

## 14.6 Ejercicios

1. Realiza una clase *Minumero* que proporcione el doble, triple y cuádruple de un número proporcionado en su constructor (realiza un método para doble, otro para triple y otro para cuádruple). Haz que la clase tenga un método *main* y comprueba los distintos métodos.
2. Crear una clase que represente una tarjeta de crédito. La clase tendrá el atributo *saldo* y tres métodos: *pagar*, *ingresar* y *verSaldo*. *Pagar* disminuye el saldo en la cantidad que se le pase, *ingresar* lo aumenta y *verSaldo* lo muestra. Si al crear el objeto no se le pasa nada como parámetro el saldo inicial será 0, sino el pasado como parámetro. Realizar también una clase que muestre el funcionamiento de la clase anterior.

3. Crear una clase que implemente una pila de números enteros positivos. El constructor pedirá el tamaño de la pila. La clase implementara un método put para añadir un número a la pila, el método get para sacarlo, el método mostrar para mostrarla y el método size para ver el tamaño actual de la pila. Usar un vector para implementar la pila).
4. Crea una clase “punto3d” que extienda la clase Point para trabajar con puntos en tres dimensiones, implementar el método move, translate y printPoint para mover el punto, trasladarlo e imprimir las coordenadas en pantalla respectivamente.(java.awt.Point).

## 15. Object

En Java cualquier clase que hagamos es una subclase de la clase *Object*, es como si las clases se crearan a partir de una “plantilla maestra”. Todos los objetos creados heredan de *Object*. De todos los métodos que tiene *Object* hay dos que son importantes para el fin de este manual.

- *toString()*: Devuelve una representación de un objeto como String.
- *equals()*: Compara dos objetos para ver si son iguales.

Veamos el primero, imaginemos el siguiente código:

### 15.1 *toString()*

```

3  public class Persona {
4      private String nombre;
5      private String apellidos;
6      private int edad;
7      private String nif;
8
9      public Persona(String nombre, String apellidos, int edad, String nif) {
10         this.nombre = nombre;
11         this.apellidos = apellidos;
12         this.edad = edad;
13         this.nif = nif;
14     }
15
16     public String getNombre() {
17         return nombre;
18     }
19
20     public void setNombre(String nombre) {
21         this.nombre = nombre;
22     }
23
24     public String getApellidos() {
25         return apellidos;
26     }
27
28     public void setApellidos(String apellidos) {
29         this.apellidos = apellidos;
30     }
31
32     public int getEdad() {
33         return edad;
34     }
35
36     public void setEdad(int edad) {
37         this.edad = edad;
38     }
39
40     public String getNif() {
41         return nif;
42     }
43
44     public void setNif(String nif) {
45         this.nif = nif;
46     }
47 }
48 
```

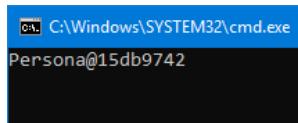
Ilustración 195. *toString*

Si quisieramos imprimir las propiedades de *Persona*, tendríamos que crear un *String* concatenando las llamadas a todos los *getters* para obtenerlas, para evitar ese trabajo existe el método *toString()*, pero ¿qué sucede si uso el método *toString* en el código anterior? Veámoslo:

```

3  public class Test {
4
5      public static void main (String[] args) {
6
7          Persona pepe=new Persona("Pepe","Gil",67,"19893476M");
8          System.out.println(pepe.toString());
9
10     }
11 } 
```

Ilustración 196. Ejemplo *toString*



¿Por qué imprime eso?, muy simple. En el *main* hemos llamado al *toString* pero en Pepe no hay ningún *toString* por lo que ha ejecutado el por defecto de la clase *Object* (ya que es su padre), dado que *Object* no sabe quién o qué es *Pepe* nos muestra los datos del Objeto, su nombre y su código hash. Para que esto no suceda, en nuestras clases debemos de sobreescribir el método *toString()* para que muestre los datos como nosotros queramos.

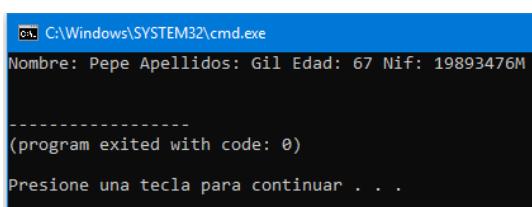
```

3  public class Persona {
4      private String nombre;
5      private String apellidos;
6      private int edad;
7      private String nif;
8
9      public Persona(String nombre, String apellidos, int edad, String nif) {
10         this.nombre = nombre;
11         this.apellidos = apellidos;
12         this.edad = edad;
13         this.nif = nif;
14     }
15
16     public String getNombre() {
17         return nombre;
18     }
19
20     public void setNombre(String nombre) {
21         this.nombre = nombre;
22     }
23
24     public String getApellidos() {
25         return apellidos;
26     }
27
28     public void setApellidos(String apellidos) {
29         this.apellidos = apellidos;
30     }
31
32     public int getEdad() {
33         return edad;
34     }
35
36     public void setEdad(int edad) {
37         this.edad = edad;
38     }
39
40     public String getNif() {
41         return nif;
42     }
43
44     public void setNif(String nif) {
45         this.nif = nif;
46     }
47     public String toString(){
48         return "Nombre: "+nombre+" Apellidos: "+apellidos+" Edad: "+edad+" Nif: "+nif;
49     }
50 }
51

```

Ilustración 197. *toString* correcto

En la línea 47 podemos ver el método, ahora Java lo encontrará y ejecutará mostrando los datos en el formato que prefiramos:



No es buena práctica no llamarle *toString()* al método que muestra todas las propiedades, primero por buena praxis y segundo porque puede que luego hayan métodos de Java que queramos utilizar y no funcionen bien dado que no encuentran el método.



*toString()* es el método predefinido para imprimir un objeto con *print* y *println*, por lo que podemos hacer perfectamente lo siguiente y obtendremos el mismo resultado: *System.out.println(pepe)*.

## 15.2 equals()

El *equals* nos va a decir si dos objetos son iguales. Veamos un ejemplo:

Definimos la clase rectángulo de esta forma

```

4  public class Rectangulo {
5
6      private int alto;
7      private int ancho;
8
9      Rectangulo(int alto,int ancho){
10         this.alto=alto;
11         this.ancho=ancho;
12     }
13 }
```

Ilustración 198. equals 1

```

4  public class Prueba {
5
6      public static void main (String[] args) {
7
8          Rectangulo r1=new Rectangulo(5,7);
9          Rectangulo r2= new Rectangulo(5,7);
10         Rectangulo r3=r1;
11         Rectangulo r4=new Rectangulo(8,9);
12         if(r1.equals(r2))
13             System.out.println("IF1:iguales: r1:"+r1.toString()+" r2:"+r2.toString());
14         else
15             System.out.println("IF1:NO iguales:+r1:"+r1.toString()+" r2:"+r2.toString());
16
17         if(r1.equals(r3))
18             System.out.println("IF2:iguales:"+r1.toString()+" r3:"+r3.toString());
19         else
20             System.out.println("IF2:NO iguales"+r1.toString()+" r3:"+r3.toString());
21
22         if(r1.equals(r4))
23             System.out.println("IF3:iguales:"+r1.toString()+" r4:"+r4.toString());
24         else
25             System.out.println("IF3:NO iguales:"+r1.toString()+" r4:"+r4.toString());
26
27     }
28 }
```

Ilustración 199. equals 2

Veamos la salida de la ejecución

```
C:\Windows\SYSTEM32\cmd.exe
IF1:NO iguales:+r1:Rectangulo@15db9742 r2:Rectangulo@6d06d69c
IF2:iguales:Rectangulo@15db9742 r3:Rectangulo@15db9742
IF3:NO iguales:Rectangulo@15db9742 r4:Rectangulo@7852e922
```

Ilustración 200. Salida equals 2

Tenemos cuatro objetos de tipo Rectangulo (r1, r2, r3 y r4). Como podemos comprobar del primer *if* *r1* y *r2* son distintos, no son el mismo objeto, aunque tengan los mismos valores. Obviamente el tercer *if* (22) también son distintos, pues no tienen ni los mismos valores, pero fijémonos en el segundo (17), los objetos son iguales, de hecho, son el mismo objeto ya que en

la línea 10 hemos hecho  $r3=r1$  y esto no hace una copia del objeto, sino que hace que  $r3$  apunte a la misma dirección de memoria donde se encuentra  $r1$ . Si cambio  $r3$ , cambio  $r1$ .

Con esto podemos ver que el *equals* se está comportando como un `==`, entonces ¿Cómo puedo hacer para que compare rectángulos por sus propiedades? Dos rectángulos son iguales si sus propiedades lo son. Bien, pues aquí entra lo visto con el *toString*, sobreescrivimos el método *equals* en la clase *Rectangulo* añadiendo el siguiente método:

```

14  @Override
15  public boolean equals(Object o){
16      if((o instanceof Rectangulo) && ((Rectangulo)o).alto==this.alto && ((Rectangulo)o).ancho==this.ancho)
17          return true;
18      else
19          return false;
20  }

```

Veamos cómo funciona:

- En la línea 14 aparece una cosa “extraña” que antes no habíamos visto pero que suele añadirse a todos los métodos sobrescritos. El `@Override` indica que es un método sobrescrito que ya existe en una clase superior, esto a parte de ayudarnos a identificarlos enseguida hace que los IDE puedan generar documentación de forma automática de nuestro código.
- La siguiente línea tiene la cabecera del método donde se define que va a devolver un booleano y que recibe un Object. Esto se hace para hacerla lo más general posible.
- En la línea 16 lo primero que se hace es comprobar si el objeto que le llega por parámetro es de tipo rectángulo, pues si no lo es no hay nada más que hacer, no son iguales. Para comprobar si un objeto es del tipo de una clase usamos `instanceof`.
- Si el objeto es un rectángulo comprobamos que su altura es la misma que la del nuestro y que su anchura también, si ambas son iguales el objeto es igual.

De esta forma nosotros decidimos cuando dos objetos son iguales, si por una propiedad, por varias, ...

Si ejecutamos el código anterior obtenemos ahora la siguiente salida:

```

C:\Windows\SYSTEM32\cmd.exe
IF1:iguales: r1:Rectangulo@15db9742 r2:Rectangulo@6d06d69c
IF2:iguales:Rectangulo@15db9742 r3:Rectangulo@15db9742
IF3:NO iguales:Rectangulo@15db9742 r4:Rectangulo@7852e922

```

Podemos comprobar como ahora  $r1$  y  $r2$  sí son iguales, y  $r4$  sigue siendo distinto, obviamente  $r3$  sigue siendo el mismo objeto que  $r1$ .

### Reglas que sigue el método equals

- **Reflexivo:** Para cualquier referencia al valor **x**, **x.equals(x)** debe regresar true.
- **Simétrico:** Para cualquier referencia a los valores **x** y **z**, **x.equals(z)** debe regresar true sí y solo sí **z.equals(x)** es true.
- **Transitivo:** Para cualquier referencia a los valores **w**, **x** y **z**, si **w.equals(x)** regresa true y **x.equals(z)** regresa true, entonces **w.equals(z)** debe regresar true.
- **Consistente:** Para cualquier referencia a los valores **x** y **z**, múltiples invocaciones a **x.equals(z)** consistentemente regresaran true o false, si es que los valores utilizados para la comparación de los objetos no ha sido modificada.
- Para cualquier **referencia no nula** del valor **x**, **x.equals(null)**, debe regresar false.

Asegúrate de sobrescribir el método *equals*. Las siguientes son implementaciones del método *equals* que son válidas para el compilador, pero no válidas para sobrescribir el método:

1. *boolean equals(Object o)*. Esta implementación no sobrescribe el método *equals* de la clase *Object*, ya que el método debe ser declarado como *public*.
2. *public boolean equals(Demo o)*. Esta implementación no sobrescribe el método *equals* de la clase *Object*, ya que el parámetro que necesita el método *equals* debe ser explícitamente un objeto de la clase *Object*, y no uno que extienda de éste. Esta implementación, al igual que la anterior es una sobrecarga del método *equals*, mas no sobrescribe este método.

La descripción correcta del método *equals*, es decir, la forma en cómo debe sobrescribirse es la siguiente (podéis comprobarlo en la API):

```
public boolean equals(Object o);
```

### Sobrescribiendo el método hashCode

#### ¿Qué implica el hashCode?

Algunas colecciones usan el valor *hashcode* para ordenar y localizar a los objetos que están contenidos dentro de ellas. El *hashcode* es un numero entero, sin signo, que sirve en colecciones de tipo Hash\* para un mejor funcionamiento en cuanto a rendimiento. Este método debe ser sobreescrito en todas las clases que sobrescriban el método *equals*, si no se quiere tener un comportamiento extraño al utilizar las colecciones de tipo Hash\* y otras clases. Si dos objetos son iguales según el método *equals* sobreescrito, estos deberían regresar el mismo *hashcode*.

Nosotros no vamos a reescribirlo ya que en los próximos manuales usaremos otros IDE que los generan automáticamente.

### 15.3 Asignación de Objetos

Cuando trabajamos con objetos estamos usando referencias, son unos punteros a direcciones de memoria. Veamos un ejemplo, imaginemos la siguiente clase Persona:

```

3  public class Persona {
4      private String nombre;
5      private String apellidos;
6      private int edad;
7      private String nif;
8
9      public Persona(String nombre, String apellidos, int edad, String nif) {
10         this.nombre = nombre;
11         this.apellidos = apellidos;
12         this.edad = edad;
13         this.nif = nif;
14     }
15
16     public void setEdad(int e){
17         edad=e;
18     }
19
20     public String toString(){
21         return "Nombre: "+nombre+" Apellidos: "+apellidos+" Edad: "+edad+" Nif: "+nif;
22     }
23
24 }
```

Ilustración 201. Asignación a Objetos

Ahora la instanciamos de la siguiente forma:

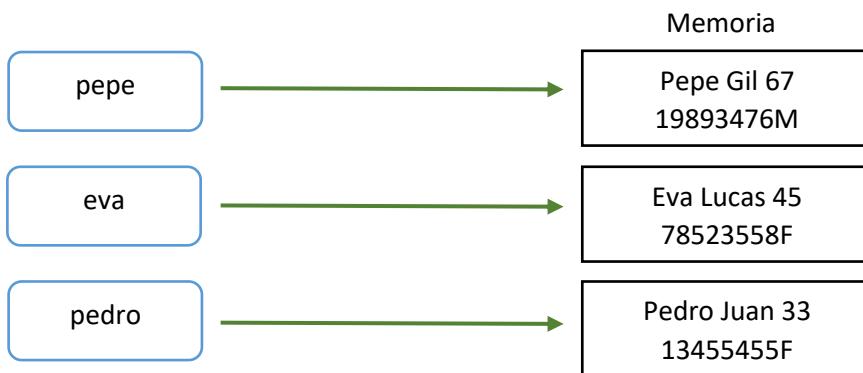
```

3  public class Test {
4
5      public static void main (String[] args) {
6
7          Persona pepe=new Persona("Pepe","Gil",67,"19893476M");
8          Persona eva=new Persona("Eva","Lucas",45,"78523558F");
9          Persona pedro=new Persona("Pedro","Juan",33,"13455455F");
10         System.out.println(pepe.toString());
11         System.out.println(pedro.toString());
12         System.out.println("-----");
13         pedro=pepe;
14         System.out.println(pepe.toString());
15         System.out.println(pedro.toString());
16         System.out.println("-----");
17         pedro.setEdad(21);
18         System.out.println(pepe.toString());
19         System.out.println(pedro.toString());
20
21     }
22 }
```

Ilustración 202. Asignación a Objetos 2

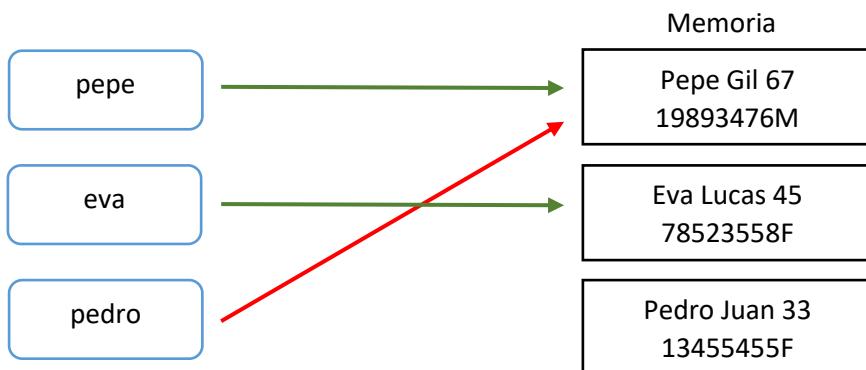
Antes de ver cuál es la salida, vamos a ver qué ocurre en memoria.

Se crean los tres objetos:



Cada objeto apunta a su propia dirección de memoria donde se encuentran sus propiedades.

Al ejecutar la línea 13, `pedro=pepe` ocurre esto:



pedro apunta a la misma dirección que pepe, por lo que son lo mismo. Si pepe cambia lo hace pedro y viceversa. ¿Qué pasa con los datos de pedro?, cuando pasa el *garbage collector* se eliminan.

Veamos el resultado de ejecutar el código anterior:

```
C:\Windows\SYSTEM32\cmd.exe
Nombre: Pepe Apellidos: Gil Edad: 67 Nif: 19893476M
Nombre: Pedro Apellidos: Juan Edad: 33 Nif: 13455455F
-----
Nombre: Pepe Apellidos: Gil Edad: 67 Nif: 19893476M
Nombre: Pepe Apellidos: Gil Edad: 67 Nif: 19893476M
-----
Nombre: Pepe Apellidos: Gil Edad: 21 Nif: 19893476M
Nombre: Pepe Apellidos: Gil Edad: 21 Nif: 19893476M
```

Ilustración 203. Salida Asignación a Objetos

Fijaros como en la línea 13 pedro pasa a ser pepe y sus datos son los mismos. Cuando le cambio la edad a pedro en la línea 17, también se la cambio a pepe.

## 15.4 Vectores de Objetos

Un vector de objetos se inicializa a *null* (como vimos, ausencia de valor). Si queremos crear un objeto en una casilla de un vector podemos hacer:

```
vector[2]=new Coche(...);
```

Por otro lado si el coche ya existe: *Coche c=new Coche(...);*

Podemos hacer:

```
vector[2]=c;
```

Para vaciar una casilla o eliminar un objeto solo tenemos que ponerlo a *null*:

```
vector[2]=null;
```



Cuidado, si hacemos *vec=null* lo que eliminamos es el vector entero.

## 15.5 Constructor copia

Visto lo anterior no podemos usar la asignación para crear una copia de un objeto, en vez de ello podemos construir lo que se llama “constructor copia”, que no es más que un constructor que recibe como parámetro un objeto de su propia clase y lo inicializa. A la clase *Persona* le añadimos el siguiente constructor:

```
15  public Persona(Persona p) {
16      this.nombre = p.nombre;
17      this.apellidos = p.apellidos;
18      this.edad = p.edad;
19      this.nif = p.nif;
20  }
```

Ilustración 204. Constructor copia

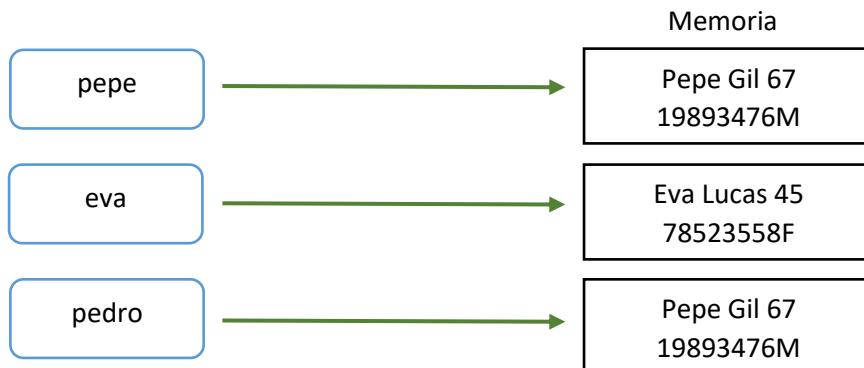
Al constructor le llega *p* y dado que es de tipo *Persona* podemos acceder a sus propiedades directamente. Ahora el programa principal lo cambiamos de esta forma:

```

3 public class Test {
4
5     public static void main (String[] args) {
6
7         Persona pepe=new Persona("Pepe","Gil",67,"19893476M");
8         Persona eva=new Persona("Eva","Lucas",45,"78523558F");
9         System.out.println(pepe.toString());
10        System.out.println("-----");
11        Persona pedro=new Persona(pepe);
12        System.out.println(pepe.toString());
13        System.out.println(pedro.toString());
14        System.out.println("-----");
15        pedro.setEdad(21);
16        System.out.println(pepe.toString());
17        System.out.println(pedro.toString());
18
19    }
20 }
```

Ilustración 205. Main constructor copia

La memoria queda así:



Como podemos ver pepe y pedro tienen los mismos datos, pero no son lo mismo. Al cambiar la edad de pedro no cambia la de pepe:

```

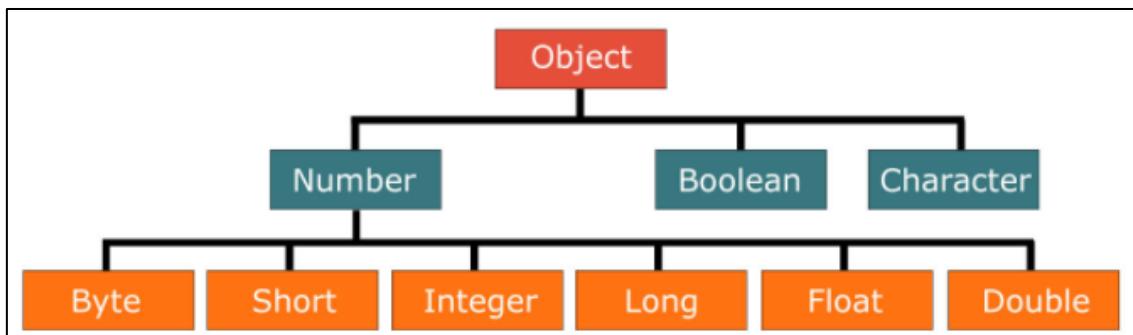
C:\Windows\SYSTEM32\cmd.exe
Nombre: Pepe Apellidos: Gil Edad: 67 Nif: 19893476M
-----
Nombre: Pepe Apellidos: Gil Edad: 67 Nif: 19893476M
Nombre: Pepe Apellidos: Gil Edad: 67 Nif: 19893476M
-----
Nombre: Pepe Apellidos: Gil Edad: 67 Nif: 19893476M
Nombre: Pepe Apellidos: Gil Edad: 21 Nif: 19893476M
```

Ilustración 206. Salida constructor copia

## 15.6 Wrappers

Un *wrapper* es una clase que representa a un tipo primitivo. A diferencia del uso de tipo primitivo son objetos por lo que se comportan como tal. Entre las ventajas de su uso se encuentra la facilidad de conversión entre tipos primitivos (suelen tener métodos *static*). Entre las desventajas, usan más memoria y hay que ir con cuidado con el paso de parámetros, siempre

se pasan por referencia. (ver tipos de paso por parámetros). Los *wrappers* se encuentran en el paquete `java.lang`, por lo que no es necesario usar `import`.



Un uso de este tipo de clases podría ser eliminar la limitación de los vectores que dice que solo pueden ser de un solo tipo. Con esto y el polimorfismo visto anteriormente los vectores pueden ser *Object* pero contener cualquier tipo de dato.

## 16. Interfaces

Una interfaz es una especie de plantilla para la construcción de clases. Normalmente una interfaz se compone de un conjunto de métodos abstractos no pudiendo tener propiedades instanciales.

Todos los métodos de una interfaz se declaran implícitamente como abstractos y públicos.

Clases abstractas vs Interfaces:

- Una clase abstracta no puede implementar los métodos declarados como abstractos, una interfaz no puede implementar ningún método (ya que todos son abstractos).
- Una clase abstracta puede tener métodos “normales” y métodos abstractos.
- Una interfaz no declara variables de instancia. Puede declarar variables finales.
- Una clase puede implementar varias interfaces, pero sólo puede tener una clase ascendiente directa.
- Una clase abstracta pertenece a una jerarquía de clases mientras que una interfaz no pertenece a una jerarquía de clases. En consecuencia, clases sin relación de herencia pueden implementar la misma interfaz.

### 16.1 Declaración de una interfaz

La declaración de una interfaz es similar a una clase, aunque emplea la palabra reservada *interface* en lugar de *class* y no incluye ni la declaración de variables de instancia ni la implementación del cuerpo de los métodos (sólo las cabeceras). La sintaxis de declaración de una interfaz es la siguiente:

```
public interface nombre {
    // Cuerpo de la interfaz ...
}
```

Una interfaz declarada como *public* debe ser definida en un archivo con el mismo nombre de la interfaz y con extensión .java. Las cabeceras de los métodos declarados en el cuerpo de la interfaz se separan entre sí por caracteres de punto y coma y todos son declarados implícitamente como *public* y *abstract*. Por su parte, todas las constantes incluidas en una interfaz se declaran implícitamente como constantes públicas y es necesario inicializarlas en la misma sentencia de declaración.

```
2 public interface Modificacion {
3     double MAXIMO = 10; //constante
4     void incremento(int a); //método abstracto
5 }
6
```

Ilustración 208. interface

## 16.2 Implementación de una interfaz en una clase

Para declarar una clase que implemente una interfaz es necesario utilizar la palabra reservada *implements* en la cabecera de declaración de la clase.

```

7 public class Acumulador implements Modificacion {
8     private int valor;
9
10    public void incremento (int a) {
11        this.valor = a;
12    }
13}

```

Ilustración 209. *implements*

## 16.3 Jerarquía entre interfaces

Se pueden implementar tantas interfaces como se quieran. Si una clase hereda de otra e implementa interfaces primero se pone la herencia y luego las implementaciones.

Otro ejemplo: pueden construirse dos interfaces: Constantes y Variaciones, y una clase, Factura, que las implementa:

```

2 public class Factura extends Documento implements Constantes, Variaciones {
3     private double totalSinIVA;
4
5     public double sinIVA() {
6         return this.totalSinIVA;
7     }
8 }

```

Ilustración 210. Herencia e implementación

Si una interfaz implementa otra, incluye todas sus constantes y declaraciones de métodos, aunque puede redefinir tanto constantes como métodos.



**Importante:** Es peligroso modificar una interfaz ya que las clases dependientes dejan de funcionar hasta que éstas implementen los nuevos métodos.

## 16.4 Interfaces como tipos

A veces creamos métodos que tenemos que estar cambiando por requerimientos del cliente. Por ejemplo imaginad un método que tenga que imprimir pero luego nos piden que salga por pantalla y luego que guarde en una BBDD. La solución pasa por usar la interface como un tipo y crear tantas clases como necesitemos usando una u otra. Veamos un ejemplo:

```

2 public interface Salida {
3     void escribe(int n);
4
5 }

```

Ilustración 211. Interfaces como tipos 1

```

2 public class Cuentas{
3     private int cuenta;
4
5     Cuentas(int a){
6         cuenta=a;
7     }
8
9     void metodoConLasCuentas (salida escritor){
10        escritor.escribe(cuenta);
11    }
12 }
```

Ilustración 212. Interfaces como tipos 2

```

2 //ejemplos de prueba
3 public class Impresora implements salida {
4
5     public void escribe(int n){
6         System.out.println("IMPRIMIENDO:"+n);
7     }
8
9
10 public class Pantalla implements salida {
11
12     public void escribe(int n){
13         System.out.println(n);
14     }
15
16
17 public class Sqlserver implements salida {
18
19     public void escribe(int n){
20         System.out.println("Guardado en la BBDD:"+n);
21     }
22 }
```

Ilustración 213. Interfaces como tipos 3

```

2 public class TestInterfaceTipo {
3
4     public static void main (String[] args) {
5         Pantalla p =new Pantalla();
6         Impresora i =new Impresora();
7         Sqlserver s=new Sqlserver();
8         Cuentas prueba=new Cuentas(120);
9         prueba.metodoConLasCuentas(p);
10        prueba.metodoConLasCuentas(i);
11        prueba.metodoConLasCuentas(s);
12
13    }
14 }
```

Ilustración 214. Interfaces como tipos 4

Fijaros como en las líneas 5, 6 y 7 se crean las clases que saben cómo imprimir, mostrar por pantalla y conectar con la BBDD. Luego se pasan como parámetro al mismo método que las puede aceptar ya que todas implementan la misma interface.

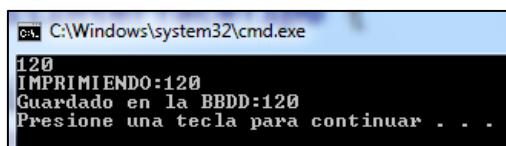


Ilustración 215. Salida Interfaces como tipos

## 16.5 Cambios en JDK8

A partir de la versión de Java 8 se introducen algunos cambios que han ido aumentando poco a poco en siguientes versiones. Los más importantes son:

### 16.5.1 Default

Si hasta ahora en una interfaz solo podíamos tener constantes y métodos abstractos ahora tendremos la posibilidad de declarar métodos con un comportamiento por defecto en una interfaz. Antes de Java 8 podíamos definir que toda clase que implementa a una interfaz debe de dar cuerpo a los distintos métodos que contenga la interfaz. En cambio, en Java 8, toda clase que implemente una interfaz debe declarar los distintos métodos que contenga la interfaz salvo aquellos que estén definidos como métodos por defecto. Estos métodos se caracterizan porque son métodos que están declarados en la propia interfaz y pueden ser utilizados directamente en la clase si nos interesa su comportamiento por defecto.

```

2 public interface Icalculadora {
3
4     int sumar(int x, int y);
5     int restar(int x, int y);
6
7     default int multiplicar(int x, int y){
8         return x*y;
9     }
10 }
```

Ilustración 216. Métodos por defecto 1

```

2 public class Calculadora implements Icalculadora {
3
4     public int sumar(int x, int y){
5         return x+y;
6     }
7
8     public int restar(int x, int y){
9         return x-y;
10    }
11 }
```

Ilustración 217. Métodos por defecto 2

```

2 public class Test {
3
4     public static void main (String args[]) {
5
6         Calculadora c=new Calculadora();
7
8         System.out.println(c.sumar(2,3));
9         System.out.println(c.restar(2,3));
10        System.out.println(c.multiplicar(2,3));
11
12    }
13 }
```

Ilustración 218. Métodos por defecto 3

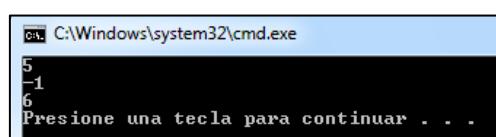


Ilustración 219. Salida Métodos por defecto

Como podemos observar hemos declarado en la interface el método *multiplicar* como *default*, como este método no está sobrescrito en la clase *Calculadora* ejecuta el código predefinido en la interface.

Si *Calculadora* soluciona el método de la interface *multiplicar* este actuará como se implemente en calculadora.

```

2  public class Calculadora implements Icalculadora {
3
4      public int sumar(int x, int y){
5          return x+y;
6      }
7
8      public int restar(int x, int y){
9          return x-y;
10     }
11
12     public int multiplicar(int x, int y){
13         return x*y/2;
14     }
15 }
```

Ilustración 220. Métodos por defecto 4

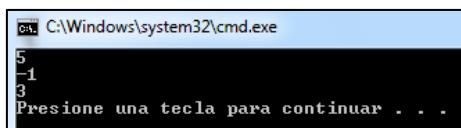


Ilustración 221. Salida Métodos por defecto 4

### 16.5.2 Interface funcional

Es una interface que tiene un único método. A partir de JDK 8 existe la anotación `@FunctionalInterface`.

### 16.5.3 Expresiones Lambda

Usando las interfaces funcionales podemos crear expresiones lambda.

Una expresión lambda es un método anónimo (su cuerpo no se ha escrito en un método a parte sino en el mismo momento de su uso).

Sintaxis: `(parámetros) -> {cuerpo lambda}`

- El operador `->` separa la declaración de parámetros de la declaración del cuerpo del método
- Parámetros
  - Cuando hay un único parámetro no es necesario usar paréntesis
  - Cuando no hay parámetros o hay más de uno es necesario el paréntesis y separarlos por comas
- Cuerpo lambda
  - Si está formado por una única línea no son necesarias las llaves ni el operador `return`, si necesita devolver algo.

- Si está formado por más de una línea es obligatorio el uso de llaves y de la palabra *return* si va a devolver algo.

## Ejemplos

Ej1. Sin necesidad de parámetros

```

2  @FunctionalInterface
3  public interface Icalculadora2 {
4
5      void mensaje();
6
7  }

```

Ilustración 222. Expresiones Lambda 1

```

2  public class Test {
3
4      public static void main (String args[]) {
5
6          Icalculadora2 calculadora=()-> System.out.println("Hola Mundo");
7
8          calculadora.mensaje();
9
10 }

```

Ilustración 223. Expresiones Lambda 2

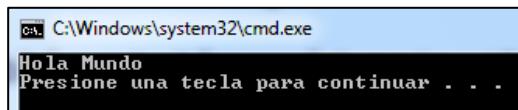


Ilustración 224. Salida Expresiones Lambda 2

Ej2. Un parámetro

```

2  @FunctionalInterface
3  public interface Icalculadora2 {
4
5      void mensaje(String a);
6
7  }

```

Ilustración 225. Expresiones Lambda 3

```

2  public class Test {
3
4      public static void main (String args[]) {
5
6          Icalculadora2 calculadora=(n)-> System.out.println(n+" Hola Mundo");
7
8          calculadora.mensaje("Soy Java.");
9
10 }

```

Ilustración 226. Expresiones Lambda 4

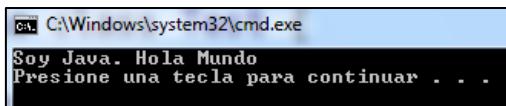


Ilustración 227. Salida Expresiones Lambda 4

Ej3. Dos parámetros, diferentes operaciones

```

2  @FunctionalInterface
3  public interface Icalculadora2 {
4
5      void operacion(double x, double y);
6
7 }
```

Ilustración 228. Expresiones Lambda 5

```

2  public class Test {
3
4      public static void main (String args[]) {
5
6          Icalculadora2 calculadora1=(n1,n2)->{
7              double resultado=n1+n2;
8              System.out.println("La suma es:"+resultado);
9          };
10         calculadora1.operacion(3,5);
11
12         Icalculadora2 calculadora2=(n1,n2)->{
13             double resultado=n1-n2;
14             System.out.println("La resta es:"+resultado);
15         };
16         calculadora2.operacion(10,6);
17
18     }
19 }
```

Ilustración 229. Expresiones Lambda 6

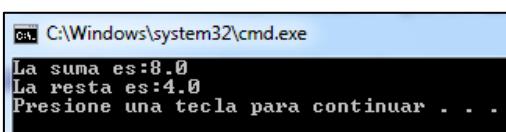


Ilustración 230. Salida Expresiones Lambda 6

## 16.6 ¿Qué es Comparable?

*Comparable* es un interfaz de java, que está creada o desarrollada dentro del JRE de java, por lo cual no es necesario importar nada, tan solo implementarla directamente en las clases con *implements Comparable* y especificar el tipo de clase (objeto). Es parametrizada o genérica.

### 16.6.1 ¿Para qué sirve Comparable?

*Comparable* contiene un método abstracto, *compareTo*, el cual permite ordenar un objeto según una propiedad o propiedades especificadas en un orden ascendente o descendente.

### 16.6.2 ¿Cómo funciona compareTo?

*CompareTo* es un método abstracto que retorna un *int* para todos los casos, n<0 si el objeto es menor, n>0 si el objeto es mayor, o n=0 si son iguales, dependiendo del objeto

existente que comparemos con el objeto de entrada, que va ser del tipo especificado en la implementación, lo sobrescribimos para cada caso.

Para implementar comparable la sintaxis es:

```
implements Comparable<T>
```

Siendo T la clase a comparar. Normalmente coincide con la clase que lo implementa.

### 16.6.3 ¿Cómo comparar dos objetos?

```
if (Obj1.compareTo(Obj2) > 0)
...
...
```

La clase Obj1 es la que llama a *compareTo*. Esa clase sobreescribe el método devolviendo un número positivo, negativo o 0 (no os compliqueis la vida: 1, -1 o 0).

Si usáis un *compareTo* de alguna clase de la API veréis que devuelve un número a veces grande, el número no importa sólo si es positivo, negativo o 0.

*iOjo, no funciona como equals! Si cambiamos el orden hay que cambiar la condición*

```
public class Persona implements Comparable<Persona>{
    public int dni, edad;
    public Persona( int d, int e){
        this.dni = d;
        this.edad = e;
    }

    public int compareTo(Persona o) {
        int resultado=0;
        if (this.edad<o.edad) { resultado = -1; }
        else if (this.edad>o.edad) { resultado = 1; }
        else {
            if (this.dni<o.dni) { resultado = -1; }
            else if (this.dni>o.dni) { resultado = 1; }
            else { resultado = 0; }
        }
        return resultado;
    }
}
```

Ilustración 231. *compareTo*

### 16.6.4 ¿Cómo llamar a compareTo?

Para invocar el método *compareTo* y lograr su correcto funcionamiento se necesitan dos objetos a comparar.

```
public class Programa {  
  
    public static void main(String arg[]) {  
        Persona p1 = new Persona(74999999,35);  
        Persona p2 = new Persona(72759474,30);  
        if (p1.compareTo(p2) < 0 ) { System.out.println("La persona p1: es menor."); }  
        else if (p1.compareTo(p2) > 0 ) {System.out.println("La persona p1: es mayor."); }  
        else { System.out.println ("La persona p1 es igual a la persona p2"); }  
    }  
}
```

Ilustración 232. Prueba compareTo

El *compareTo* usa toda su potencia cuando se comparan colecciones<sup>11</sup>.

#### 16.6.5 Ejercicio

- Realizar un programa que cree un vector de 10 manzanas. Las manzanas tienen un color, variedad y peso. El color puede ser “verde”, “rojo” o “amarillo”. Las manzanas se introducirán una a una, mezclando colores, pesos y variedades, en el vector. Una vez creado el vector mostrar las propiedades de cada manzana por pantalla. Ordenar el vector usando el método de la burbuja y luego volver a mostrar por pantalla la información. Para ordenar el vector se tendrá en cuenta que primero van las manzanas verdes de menor a mayor peso, luego las rojas de menor a mayor peso y al final las amarillas de menor a mayor peso.

Crear las clases necesarias. Pensad donde va el vector.

---

<sup>11</sup> Estructuras con más de un objeto. Vectores.

## 17. La clase String

Llevamos usando desde el principio el tipo *String* que sirve para tratar con cadenas de caracteres. Al principio os comenté que dábamos un salto de fe y nos creímos que no es un tipo de dato primitivo pero que, dado que es tan usado, Java lo trata muchas veces como tal. Bien, ahora podemos entender lo que es, un *String* no es más que una clase, que se encuentra en *java.lang*, por eso no hace falta importarla, y que nos permite crear objetos para trabajar con texto. Como toda clase tiene unos constructores, pero como hemos visto Java nos permite declarar un objeto de tipo *String* sin usar el operador new asignándole un valor.

Veamos un ejemplo de los métodos más usados. Podéis consultarlos todos en la API de Java.

- **Constructores:**

- `String cadena1="Hola";`
- `String cadena2= new String("Hola");`
- `String cadena3=new String(cadena2);`

- **int length()**

Devuelve la longitud de la cadena

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

Muestra 4

- **String concat(String s)**

Concatena dos String. Como el operador +

- **String toString()**

Devuelve la propia cadena

- **String trim()**

Devuelve una cadena eliminando los espacios por delante y por detrás

```
String cadena="      hola como estás      ";
System.out.println(cadena.trim());
```

Mostrará "hola como estás"

- **String toLowerCase() y String toUpperCase()**

Devuelven la cadena en minúsculas o mayúsculas respectivamente

```
String cadena="Hola como estás";
System.out.println(cadena.toUpperCase());
```

Mostrará "HOLA COMO ESTÁS"

- **String replace(char c, char newc);**

Reemplaza cada ocurrencia del carácter c por newc

```
String cadena="hola como estás";
String cadena2=cadena.replace('o','a');
```

En cadena2 hay "hala cama estás"

- **String substring(int i, int f)**

Devuelve una subcadena que comienza por el carácter i y termina en el f-1. Si no se especifica f devuelve hasta el final

```
String cadena="Juan Carlos Moreno";
System.out.println(cadena.substring(5,11));
System.out.println(cadena.substring(12));
```

La primera devuelve "Carlos" y la segunda "Moreno"

- **boolean startsWith(String s) y boolean endsWith(String s)**

Devuelven true o false si la cadena comienza por s o termina por s respectivamente.

- **char charAt(i)**

Devuelve el carácter que se encuentra en la posición i

```
System.out.println("hola".charAt(1));
```

Muestra 'o'

- **int indexOf(String s)**

Devuelve el índice donde se encuentra la primera subcadena de s o -1 si no se encuentra.

- **String valueOf(int n)**

Convierte un número entero a String

- **int compareTo(String s)**

Este es un método que se usa en muchos tipos de objetos (ya veremos cómo). Sirve para comparar dos objetos, en nuestro caso String's, devuelve:

- Un número <0 si el String llamante es menor que el pasado por parámetro.
- Un número >0 si el String llamante es mayor que el pasado por parámetro.
- 0 Si son iguales.

Para saber si es mayor o menor el método compara los String carácter a carácter de izquierda a derecha según el alfabeto. Por ejemplo: la a es menor que la d.

```
String cadena1="casa";
String cadena2="casi";
System.out.println(cadena1.compareTo(cadena2));
```

El llamante es cadena1, son iguales hasta la última 'a'<'i'---->menor que 0.



- El método `compareTo()` distingue entre mayúsculas y minúsculas, las mayúsculas se encuentran antes que las minúsculas por lo que 'A' es menor que 'a'
- No importa el número en sí que devuelve, solo si es mayor o menor que 0, salvo si es justamente 0 que significa que son iguales

- **boolean equals(String s)**

Como ya hemos visto con anterioridad, este método sirve para comprobar si dos objetos son iguales, en este caso dos *String*. En caso de ser iguales el llamante que el parámetro devuelve true, si no false.

```
String cadena1="casa";
String cadena2="casi";
System.out.println(cadena1.equals(cadena2));
```

Devuelve false.



**Los String debemos de compararlos siempre usando el método equals() y no ==.**

```
Scanner sc=new Scanner(System.in);
String cadena1="casa";
String cadena2=sc.nextLine();
System.out.print(cadena1.equals(cadena2));
System.out.print(cadena1==cadena2);
```

Si introducimos por teclado la palabra "casa", la salida de este código será: true false  
Esto es debido a que el operador == está comparando la referencia de los objetos, no su contenido. Hay una excepción, que no voy a mencionar para no liaros. Como regla los String SIEMPRE se comparan con equals()

## 18. La clase Math

La clase *Math* es una clase para realizar operaciones matemáticas que se encuentra en *java.lang* por lo que no hace falta importarla. Esta clase es estática lo que significa que todos sus métodos son estáticos y no se necesita instanciar, para usar un método simplemente anteponemos el nombre de la clase. Entre los métodos más importantes que podemos encontrar están:

<code>double random()</code>	Número aleatorio de 0 a 1
<code>tipo abs(tipo x)</code>	Devuelve el valor absoluto de x.
<code>double sqrt(double x)</code>	Calcula la raíz cuadrada de x
<code>double pow(double x,double y)</code>	Calcula x elevado a y
<code>double exp(double x)</code>	Calcula $e^x$

Tabla 26. Clase Math

Si miramos la API podemos ver como hay muchos más:

<code>static int decrementExact(int a)</code>	Returns the argument decremented by one, throwing an exception if the result overflows an int.
<code>static long decrementExact(long a)</code>	Returns the argument decremented by one, throwing an exception if the result overflows a long.
<code>static double exp(double a)</code>	Returns Euler's number $e$ raised to the power of a double value.
<code>static double expm1(double x)</code>	Returns $e^x - 1$ .
<code>static double floor(double a)</code>	Returns the largest (closest to positive infinity) double value that is less than or equal to the argument and is equal to a mathematical integer.
<code>static int floorDiv(int x, int y)</code>	Returns the largest (closest to positive infinity) int value that is less than or equal to the algebraic quotient.
<code>static long floorDiv(long x, long y)</code>	Returns the largest (closest to positive infinity) long value that is less than or equal to the algebraic quotient.
<code>static int floorMod(int x, int y)</code>	Returns the floor modulus of the int arguments.
<code>static long floorMod(long x, long y)</code>	Returns the floor modulus of the long arguments.
<code>static int getExponent(double d)</code>	Returns the unbiased exponent used in the representation of a double.
<code>static int getExponent(float f)</code>	Returns the unbiased exponent used in the representation of a float.
<code>static double hypot(double x, double y)</code>	Returns $\sqrt{x^2 + y^2}$ without intermediate overflow or underflow.
<code>static double IEEEremainder(double f1, double f2)</code>	Computes the remainder operation on two arguments as prescribed by the IEEE 754 standard.
<code>static int incrementExact(int a)</code>	Returns the argument incremented by one, throwing an exception if the result overflows an int.
<code>static long incrementExact(long a)</code>	Returns the argument incremented by one, throwing an exception if the result overflows a long.
<code>static double log(double a)</code>	Returns the natural logarithm (base $e$ ) of a double value.
<code>static double log10(double a)</code>	Returns the base 10 logarithm of a double value.

Ilustración 233. Extracto de métodos Math

Como podéis observar todas son *static*. Además, también se declaran dos constantes: PI y E.

Veamos un ejemplo de uso de *Math*:

```
3 public class Pruebas {
4
5     public static void main (String[] args) {
6         int x=5;
7         int y=-7;
8         double z=5.8;
9
10        System.out.println(Math.PI*Math.pow(5,2)); //radio de una circunferencia
11        System.out.println(Math.abs(y));
12        System.out.println(Math.cos(x));
13        System.out.println(Math.sqrt(x));
14        System.out.println(Math.round(z));
15
16    }
17 }
```

Ilustración 234. Ejemplo Math

```
C:\Windows\SYSTEM32\cmd.exe
78.53981633974483
7
0.28366218546322625
2.23606797749979
6

-----
(program exited with code: 0)
```

Ilustración 235. Salida Math

## 19. Anexo I. ArrayList

La clase *ArrayList*, es una clase que implementa la interface *List* y que permite almacenar datos en memoria de forma similar a los vectores, con la ventaja de que el número de elementos que almacena es dinámico (en tiempo de ejecución), es decir, que no es necesario declarar su tamaño como pasa con los vectores. Además, permite cambiar su tamaño dinámicamente conforme se insertan o eliminan elementos.

*ArrayList* se encuentra en el paquete *java.util*, por tanto hay que importarla para poder usarla.

Si nos fijamos en su definición podemos ver lo siguiente:

```
Class ArrayList<E>
    java.lang.Object
        java.util.AbstractCollection<E>
            java.util.AbstractList<E>
                java.util.ArrayList<E>
```

La declaración *<E>* nos indica que es una clase genérica o parametrizada, este tipo de clases permiten realizar operaciones sea cual sea el tipo que reciban. En este caso la *E* significa *elemento de una colección*.

Las clases genéricas tienen algunas restricciones:

- **No se pueden instanciar tipos genéricos con tipos primitivos.**
- No se pueden crear instancias de los parámetros de tipo.
- No se pueden declarar campos *static* cuyos tipos son parámetros de tipo.
- No se pueden usar castings o *instanceof* con tipos parametrizados.
- No se pueden crear *arrays* de tipos parametrizados.
- No se pueden crear, capturar o lanzar tipos parametrizados que extiendan de *Throwable*.

Veamos el uso de *ArrayList* a través de distintas aproximaciones

```
1 import java.util.ArrayList;
2
3 public class lista1 {
4
5     public static void main(String[] args) {
6         ArrayList<String> listado = new ArrayList<String>();
7
8         listado.add("Elemento1");
9         listado.add("Elemento2");
10
11        for (int i = 0; i < listado.size(); i++) {
12            System.out.println(listado.get(i));
13        }
14    }
15 }
16 }
```

Ilustración 236. Ejemplo ArrayList 1

Como podemos ver, lo primero que debemos de hacer es importarlo.

Para declararlo usamos la siguiente sintaxis:

```
ArrayList<Tipo> listado = new ArrayList<Tipo>();
```

Donde *Tipo* es la clase a la cual pertenecen los objetos que se van a guardar. Termina con () ya que estamos llamando al constructor por defecto.



A partir de la versión 7 del JDK no es necesario declarar el tipo en la parte de la derecha.

```
ArrayList<Tipo> listado = new ArrayList<>();
```

Dado que no es un vector no podemos usar [], debemos de usar el método *add* para añadir objetos. Los *ArrayList*, al igual que los vectores, comienzan por el índice 0.

Podemos ver un resumen de los métodos en el siguiente cuadro:

MÉTODOS	DESCRIPCIÓN
<i>add(Object)</i>	Agrega un elemento al final. <pre><code>public void adicionar(Producto x) {     prod.add(x); }</code></pre>
<i>add(int, Object)</i>	Agrega un elemento en la posición especificada en el primer parámetro. <pre><code>prod.add(0,x);</code></pre>
<i>clear()</i>	Elimina todos los elementos. <pre><code>prod.clear();</code></pre>
<i>get(int)</i>	Devuelve el elemento de la posición especificada. <pre><code>public Producto obtener(int pos) {     return prod.get(pos); }</code></pre>
<i>indexOf(Object)</i>	Devuelve el índice del elemento especificado, de no encontrarlo devuelve -1. <pre><code>public int posicion(Producto x) {     return prod.indexOf(x); }</code></pre>
<i>remove(int)</i>	Elimina el elemento de la posición especificada. <pre><code>public void eliminar(int x) {     prod.remove(x); }</code></pre>
<i>remove(Object)</i>	Elimina el elemento especificado. <pre><code>public void eliminar(Producto x) {     prod.remove(x); }</code></pre>
<i>set(int, Object)</i>	Reemplaza el elemento de la posición especificada en el primer parámetro por el elemento del segundo parámetro. <pre><code>public void modificar(int pos, Producto x) {     prod.set(pos,x); }</code></pre>
<i>size()</i>	Devuelve la cantidad de elementos agregados. <pre><code>public int tamaño(){     return prod.size(); }</code></pre>

Tabla 27. Métodos *ArrayList*

También podemos usar el constructor con el que le damos un tamaño inicial:

```
ArrayList<Tipo> listado = new ArrayList<Tipo>(int n);
```

Pero, ¿Por qué usar este constructor si el *ArrayList* puede crecer dinámicamente? Muy simple. Cuando se crea un *ArrayList* sin especificar el tamaño, este se crea de forma predeterminada para diez elementos, cuando insertamos uno más, internamente, Java lo que

hace es realizar una copia del *ArrayList* en otro de tamaño once y añadir el siguiente elemento y así sucesivamente, pero esto tiene un coste. Si desde el principio le indicamos el tamaño que creemos más acertado para el uso que le vamos a dar, el coste es menor.

Para obtener o cambiar un elemento usamos el método set y get:

```

1 import java.util.ArrayList;
2
3 public class Listal {
4
5     public static void main(String[] args) {
6         ArrayList<String> listado = new ArrayList<String>(15);
7
8         listado.add("Elemento1");
9         listado.add("Elemento2");
10        listado.set(1,"Elemento3");
11
12        for (int i = 0; i < listado.size(); i++) {
13            System.out.println(listado.get(i));
14        }
15    }
16
17 }
```

Ilustración 237. Ejemplo *ArrayList* 2

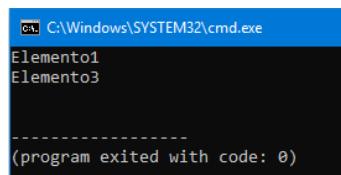


Ilustración 238. Salida Ejemplo *ArrayList* 2

También podemos insertar elementos en una posición dada sin borrar el que hay en ella y eliminar el de una posición dada, en ambos casos el *ArrayList* se redimensiona sin perder la información.

```

4     public static void main(String[] args) {
5         ArrayList<String> listado = new ArrayList<>();
6
7         listado.add("Elemento1");
8         listado.add("Elemento2");
9         listado.add("Elemento3");
10        listado.add("Elemento4");
11        listado.add(1,"ElementoNuevo");
12
13        for (int i = 0; i < listado.size(); i++) {
14            System.out.println(listado.get(i));
15        }
16        System.out.println("*****");
17        listado.remove(2);
18        for (int i = 0; i < listado.size(); i++) {
19            System.out.println(listado.get(i));
20        }
21    }
```

Ilustración 239. Ejemplo *ArrayList* 3

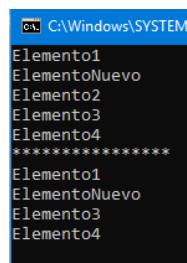


Ilustración 240. Salida Ejemplo *ArrayList*

Ejemplo de otros métodos de ArrayList (Ver API):

```
public static void main(String[] args) {
    ArrayList<String> listado = new ArrayList<>();

    listado.add("Elemento1");
    listado.add("Elemento2");
    listado.add(1,"Elemento");

    for (int i = 0; i < listado.size(); i++) {
        System.out.println(listado.get(i));
    }

    System.out.println(listado.contains("Elemento"));
    System.out.println(listado.indexOf("Elemento"));
    listado.remove(1);
    for (int i = 0; i < listado.size(); i++) {
        System.out.println(listado.get(i));
    }

    listado.clear();
    System.out.println(listado.isEmpty());
    System.out.println(listado.size());
}
```

Ilustración 241. Ejemplo ArrayList 4

```
C:\Windows
Elemento1
Elemento
Elemento2
true
1
Elemento1
Elemento2
true
0
```

Ilustración 242. Salida Ejemplo ArrayList 4

## 19.1 Estructuras para recorrer colecciones

**For Each:** El ciclo for-each es una herramienta muy útil cuando tenemos que realizar recorridos completos de colecciones. Para recorrer un *ArrayList* de una forma más cómoda podemos usar el bucle for-each de Java cuya sintaxis es la siguiente:

```
for (String nombre:listado)
```

Este *for* recorre toda la colección (*ArrayList*), define una variable que va a tomar el valor de todos los elementos de la colección:

```
public static void main(String[] args) {
    ArrayList<String> listado = new ArrayList<>();

    listado.add("Elemento1");
    listado.add("Elemento2");
    listado.add("Elemento3");
    listado.add("Elemento4");
    listado.add(1,"ElementoNuevo");

    for (String nombre:listado) {
        System.out.println(nombre);
    }
}
```

Ilustración 243. Ejemplo for each

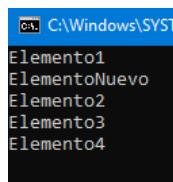


Ilustración 244. Salida Ejemplo for each

La variable *nombre* va tomando todos los valores del *ArrayList* listado.



Obviamente podemos usar el *for* de siempre que ya conocemos.

**Usando un iterador:** En Java, la interface *Iterator* provee un mecanismo estándar para acceder secuencialmente a los elementos de una colección, define una interface que declara métodos para acceder secuencialmente a los objetos de una colección. Una clase accede a una colección a través de dicha interface.

Se utiliza cuando:

- Una clase necesita acceder al contenido de una colección sin llegar a ser dependiente de la clase que es utilizada para implementar la colección, es decir sin tener que exponer su representación interna.
- Una clase necesita un modo uniforme de acceder al contenido de varias colecciones.
- Cuando se necesita soportar múltiples recorridos de una colección.

```
public interface Iterator<E>
```

La clase *ArrayList* tiene un método *iterator()* que me devuelve un iterador sobre dicha clase y que tiene una serie de métodos:

<code>Iterator&lt;E&gt;</code>	<code>iterator()</code>
Returns an iterator over the elements in this list in proper sequence.	

Los tres principales métodos de esta interface son:

<code>boolean hasNext()</code>	Returns true if the iteration has more elements.
<code>E next()</code>	Returns the next element in the iteration.
<code>default void remove()</code>	Removes from the underlying collection the last element returned by this iterator (optional operation).

- **hasNext():** me dice si hay más elementos que recorrer.
- **next():** devuelve el siguiente elemento.
- **remove():** borrar el elemento de la colección.

## Ejemplo de uso:

```

public static void main(String[] args) {
    ArrayList<String> listado = new ArrayList<>();

    listado.add("Elemento1");
    listado.add("Elemento2");
    listado.add(1,"Elemento");
    listado.add("Elemento3");
    listado.add("Elemento4");
    listado.add("Elemento5");

    Iterator<String> iterador = listado.iterator();
    while(iterador.hasNext()){
        String elemento = iterador.next();
        System.out.println(elemento);
    }
}

```

Ilustración 245. Ejemplo de iterador

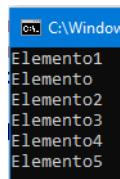


Ilustración 246. Salida Ejemplo de iterador

Veamos un ejemplo en el que tenemos una serie de personas que tienen una edad y queremos mostrar aquellas que están jubiladas:

```

1  public class Persona {
2      private String nombre;
3      private String apellidos;
4      private int edad;
5
6      public Persona(String nombre, String apellidos, int edad) {
7          this.nombre = nombre;
8          this.apellidos = apellidos;
9          this.edad = edad;
10     }
11
12     public String getNombre() {
13         return nombre;
14     }
15
16     public void setNombre(String nombre) {
17         this.nombre = nombre;
18     }
19
20     public String getApellidos() {
21         return apellidos;
22     }
23
24     public void setApellidos(String apellidos) {
25         this.apellidos = apellidos;
26     }
27
28     public int getEdad() {
29         return edad;
30     }
31
32     public void setEdad(int edad) {
33         this.edad = edad;
34     }
35
36     public boolean jubilado(int e){
37         if(edad>=e)
38             return true;
39         else
40             return false;
41     }
42
43     @Override
44     public String toString() {
45         return "Persona{" + "nombre=" + nombre + ", apellidos=" + apellidos + ", edad=" + edad + '}';
46     }
47 }

```

Ilustración 247. Ejemplo ArrayList 5

```

2 import java.util.ArrayList;
3 public class Test {
4
5     public static final int JUBILACION=67;
6     public static void main(String[] args) {
7
8         ArrayList<Persona>personas=new ArrayList<>();
9
10        personas.add(new Persona("Pepe","Gozalbo",30));
11        personas.add(new Persona("Eva","Chico",78));
12        personas.add(new Persona("Pedro","Suarez",67));
13        personas.add(new Persona("José","Sanchez",56));
14        personas.add(new Persona("Laura","Pendras",89));
15        personas.add(new Persona("Rosana","Tena",64));
16
17        for(Persona p:personas)
18            if(p.jubilado(JUBILACION))
19                System.out.println(p);
20
21    }
22 }
23

```

Ilustración 248. Test Ejemplo ArrayList 5

```

C:\Windows\SYSTEM32\cmd.exe
Persona{nombree=Eva, apellidos=Chico, edad=78}
Persona{nombree=Pedro, apellidos=Suarez, edad=67}
Persona{nombree=Laura, apellidos=Pendras, edad=89}

-----
(program exited with code: 0)

Presione una tecla para continuar . .

```

Ilustración 249. Salida Ejemplo ArrayList 5

Dado que un *ArrayList* no permite el uso de tipos primitivos, debemos de usar sus correspondientes clases o *wrappers*:

Tipos primitivos (no son objetos y por tanto no poseen métodos)	Wrappers (son objeto y por tanto poseen métodos)
byte	Byte
short	Short
int	Integer
long	Long
boolean	Boolean
float	Float
double	Double
char	Character

Ilustración 250. wrappers

## 20. Anexo II. Excepciones

Una excepción es un evento que ocurre durante la ejecución del programa que interrumpe el flujo normal de las sentencias. Muchas clases de errores pueden generar excepciones, desde problemas de hardware, como la avería de un disco duro, a los simples errores de programación, como tratar de acceder a un elemento de un vector fuera de sus límites.

Como vimos en su momento, se puede generar una excepción si el usuario escribe algo que no es un numero cuando Java espera un número:

```
C:\Windows\system32\cmd.exe
Introduce un número decimal: 456gf
Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Scanner.java:864)
    at java.util.Scanner.next(Scanner.java:1485)
    at java.util.Scanner.nextDouble(Scanner.java:2413)
    at Pruebas.main(Pruebas.java:11)
Presione una tecla para continuar . . .
```

Ilustración 251. Ejemplo de excepción

Las excepciones hay que tratarlas para que el programa no termine bruscamente y pueda continuar. Java tiene la siguiente jerarquía en cuanto a las clases que tratan las excepciones:

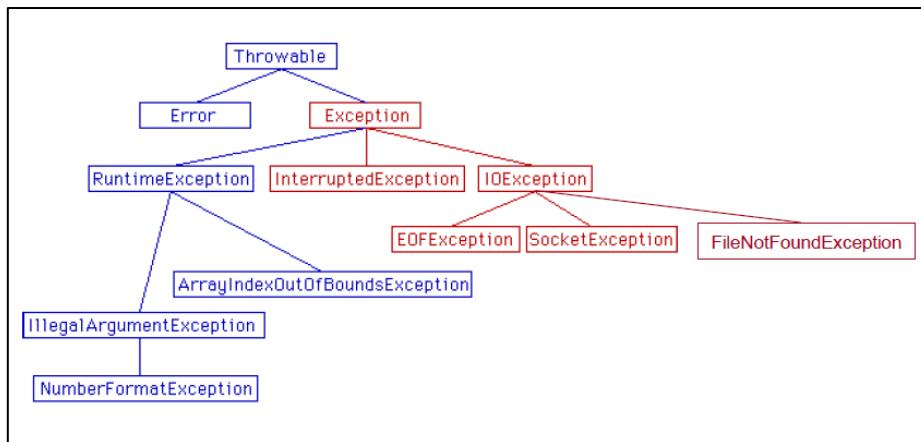


Ilustración 252. Jerarquía de excepciones

Aquellas que están en rojo estamos obligadas a tratarlas o simplemente ni siquiera compilará. Las azules no estamos obligados a tratarlas. Fijaros como las que nos han salido en este manual son las azules.

Java nos permite hacer un control de las excepciones para que nuestro programa no se pare inesperadamente, aunque se produzca una excepción. Para ello tenemos la estructura "try - catch - finally":

```

try{
    <instrucciones que hay que controlar>
}catch (typeException e){
    <instrucciones a ejecutar cuando se produce la excepción
} finally{
    <instrucciones que se ejecutan tanto si hay excepción como si no>
}
  
```

Un *try* puede tener más de un *catch* para hacer una cosa u otra dependiendo del tipo de excepción. La cláusula *finally* es optativa.

Cuando en Java se produce una excepción se crea un objeto de una determina clase (dependiendo del tipo de error que se haya producido), que mantendrá la información sobre el error producido y nos proporcionará los métodos necesarios para obtener dicha información.

Veamos un ejemplo:

```

1 public class Pruebas {
2     public static void main(String[] args) {
3
4         int x=2;
5         int y=0;
6
7         System.out.println("El resultado de dividirlos es: "+x/y);
8
9     }
10 }
```

Ilustración 253. Excepciones 1

C:\Windows\SYSTEM32\cmd.exe  
Exception in thread "main" java.lang.ArithmeticsException: / by zero  
at Pruebas.main(Pruebas.java:7)

Ilustración 254. Salida Excepciones 1

Cuando lo compilamos no da ningún error pero al ejecutarlo salta una excepción, concretamente la *ArithmeticsException* ya que no se puede dividir por 0.

La forma de capturar esta excepción sería:

```

1 public class Pruebas {
2     public static void main(String[] args) {
3
4         int x=2;
5         int y=0;
6         try{
7             System.out.println("El resultado de dividirlos es: "+x/y);
8         }
9         catch(Exception e){
10             System.out.println("ERROR: no se puede dividir por 0");
11         }
12
13     }
14
15 }
16 }
```

Ilustración 255. Excepciones 2

C:\Windows\SYSTEM32\cmd.exe  
ERROR: no se puede dividir por 0  
-----  
(program exited with code: 0)

Ilustración 256. Salida excepciones 2

Podemos usar los métodos del objeto de la excepción para obtener información:

```
1 public class Pruebas {
2     public static void main(String[] args) {
3
4         int x=2;
5         int y=0;
6         try{
7             System.out.println("El resultado de dividirlos es: "+x/y);
8         }
9         catch(Exception e){
10            System.out.println(e.getMessage());
11            System.out.println(e.getStackTrace());
12            System.out.println(e.getCause());
13        }
14    }
15 }
16 }
```

Ilustración 257. Excepciones 3

Capturar una excepción es más costoso que evitarla. En este ejemplo, al igual que en todos los de este manual, las excepciones se pueden evitar. Basta comprobar la y con un *if* antes de hacer la división. Obviamente hay muchas veces que son inevitables, pero aquellas que son fácilmente evitables hagámoslo. Hay gente que programa “orientado a excepciones”, en todo usa excepciones, personalmente no me parece la mejor forma.

Obviamente del tema de excepciones me dejo muchas cosas (que Java las ignore, personalización de excepciones, ...), eso lo dejo para futuros manuales de mayor dificultad.

## 21. Anexo III. Ordenar objetos de una colección cuando hay nulos

Hemos visto en el punto 16.6 que es el *compareTo* de la interface *Comparable* y como lo podemos usar para ordenar objetos por los parámetros que queramos. Bien, el *compareTo* funciona muy bien excepto cuando en el vector o colección de objetos existen nulos. Por ejemplo: si tengo un vector de personas e intento ordenarlo con *Arrays.sort()*, si en ese vector hay alguna casilla que no tenga una persona, que esté vacía (null) el método provocará una excepción *NullPointerException*. Vamos a ver como poder ordenar colecciones cuando esto sucede.

Veamos primero el problema:

```

3  public class Persona implements Comparable<Persona>{
4
5      private String nombre;
6      private int edad;
7
8      Persona(String nombre, int edad){
9          this.nombre=nombre;
10         this.edad=edad;
11     }
12
13     public int compareTo(Persona p){
14         if(edad>p.edad)
15             return 1;
16         else if(edad<p.edad)
17             return -1;
18         else
19             return 0;
20     }
21
22
23
24     public String toString(){
25         return "Nombre:"+nombre+", Edad:"+edad;
26     }
27 }
```

Ilustración 258. Comparator 1

```

2 import java.util.Arrays;
3 public class Test {
4
5     public static void main (String[] args) {
6
7         Persona [] vec=new Persona[5];
8         vec[0]=new Persona("Jose",43);
9         vec[1]=new Persona("Pedro",35);
10        vec[2]=new Persona("Eva",25);
11        vec[3]=new Persona("Pablo",58);
12        vec[4]=new Persona("Andrea",10);
13
14        for(int i=0;i<vec.length;i++)
15            System.out.println(vec[i]);
16
17        Arrays.sort(vec);
18
19        System.out.println("-----");
20
21        for(int i=0;i<vec.length;i++)
22            System.out.println(vec[i]);
23
24    }
25 }
```

Ilustración 259. Comparator 2

```
C:\Windows\SYSTEM32\cmd.exe
Nombre:Jose, Edad:43
Nombre:Pedro, Edad:35
Nombre:Eva, Edad:25
Nombre:Pablo, Edad:58
Nombre:Andrea, Edad:10
-----
Nombre:Andrea, Edad:10
Nombre:Eva, Edad:25
Nombre:Pedro, Edad:35
Nombre:Jose, Edad:43
Nombre:Pablo, Edad:58
```

Ilustración 260. Comparator salida 1

Como podemos ver todo ha funcionado bien, pero veamos que sucede si el vector no está lleno:

```
2 import java.util.Arrays;
3 public class Test {
4
5     public static void main (String[] args) {
6
7         Persona [] vec=new Persona[5];
8         vec[0]=new Persona("Jose",43);
9         vec[1]=new Persona("Pedro",35);
10        vec[3]=new Persona("Pablo",58);
11        vec[4]=new Persona("Andrea",10);
12
13        for(int i=0;i<vec.length;i++)
14            System.out.println(vec[i]);
15
16        Arrays.sort(vec);
17
18        System.out.println("-----");
19
20        for(int i=0;i<vec.length;i++)
21            System.out.println(vec[i]);
22    }
23
24 }
```

Ilustración 261. Comparator 3

```
C:\Windows\SYSTEM32\cmd.exe
Nombre:Jose, Edad:43
Nombre:Pedro, Edad:35
null
Nombre:Pablo, Edad:58
Nombre:Andrea, Edad:10
Exception in thread "main" java.lang.NullPointerException
at java.util.ComparableTimSort.countRunAndMakeAscending(Unknown Source)
at java.util.ComparableTimSort.sort(Unknown Source)
at java.util.Arrays.sort(Unknown Source)
at Test.main(Test.java:16)
```

Ilustración 262. Salida 2 Comparator

El método `sort` al encontrar un `null` provoca una excepción ya que no puede aplicar `null.compareTo()`.

Para solucionar el problema vamos a usar un comparador. Crearemos una clase especial para comparar `Personas` y usaremos unos parámetros distintos al llamar a `sort`. Veamos:

Creamos la clase que nos va a ayudar a comparar y que implementa la interface Comparator:

```

2 import java.util.Comparator;
3 public class ComparatorPersona implements Comparator<Persona> {
4
5     public int compare(Persona p1, Persona p2){
6         if(p1==null)
7             return 1;
8         if(p2==null)
9             return -1;
10        else
11            return p1.compareTo(p2);
12    }
13 }
14
15 }
16 }
```

Ilustración 263. Comparator 4

Como podemos observar en la línea 3 le digo que va a comparar objetos de tipo *Persona*. El método *compare* a diferencia de *compareTo* necesita los dos objetos a comparar. Primero compruebo que tengo que devolver en caso de *null* y si no son *null* llamo al *compareTo* de *Persona* que es el mismo que habíamos hecho. De esta forma consigo llevar a los *null* al final de la colección y quedando ordenada:

```

7 Persona [] vec=new Persona[5];
8 vec[0]=new Persona("Jose",43);
9 vec[1]=new Persona("Pedro",35);
10 vec[3]=new Persona("Pablo",58);
11 vec[4]=new Persona("Andrea",10);
12
13 for(int i=0;i<vec.length;i++)
14     System.out.println(vec[i]);
15
16 Arrays.sort(vec,new ComparatorPersona());
17 }
```

Ilustración 264. Comparator 5

```
C:\Windows\SYSTEM32\cmd.exe
Nombre:Jose, Edad:43
Nombre:Pedro, Edad:35
null
Nombre:Pablo, Edad:58
Nombre:Andrea, Edad:10
-----
Nombre:Andrea, Edad:10
Nombre:Pedro, Edad:35
Nombre:Jose, Edad:43
Nombre:Pablo, Edad:58
null
```

Ilustración 265. Salida 3 Comparator

## 22. Anexo IV. Algebra de Boole

El álgebra de Boole son una serie tiene una serie de propiedades que nos pueden ser útiles conocer para cambiar el formato de condiciones.

Si definimos las siguientes operaciones: el . como multiplicación equivalente a un AND ( $\&\&$ ), el + como suma equivalente a un OR ( $\|$ ) y ' equivalente a NOT ( $!$ ), junto con el conjunto de elementos  $B=\{0,1\}$  podemos decir que forman un algebra de Boole si se cumple que:

1. Propiedad conmutativa:  $A + B = B + A$  y  $A \cdot B = B \cdot A$
2. Propiedad distributiva:  $A \cdot (B+C) = A \cdot B + A \cdot C$  y  $A + B \cdot C = (A+B) \cdot (A+C)$
3. Elementos neutros diferentes:  $A + 0 = A$  y  $A \cdot 1 = A$
4. Siempre existe el complemento de a, denominado  $a'$ :  $A + A' = 1$  y  $A \cdot A' = 0$

Las condiciones vistas en el *if*, *while* y *do* cumplen estas propiedades, por tanto podemos afirmar los siguientes teoremas:

- Teorema 1: el elemento complemento  $A'$  es único. Dado una variable booleana a sólo existe un  $!a$ .
- Teorema 2 (elementos nulos): para cada elemento de B se verifica:
  - $A+1 = 1 \rightarrow$  una variable booleana a  $\| 1$  siempre da 1
  - $A \cdot 0 = 0 \rightarrow$  una variable booleana a  $\&\& 0$  siempre da 0
- Teorema 3: cada elemento identidad es el complemento del otro:
  - $0'=1 \rightarrow$  dada una variable booleana a=false  $!a$  siempre da true.
  - $1'=0 \rightarrow$  dada una variable booleana a=true  $!a$  siempre da false.
- Teorema 5 (involución): para cada elemento de B, se verifica:
  - $(A')' = A \rightarrow$  dada una variable booleana a  $!(!a)=a$ .
- Teorema 6 (absorción): para cada par de elementos de B, se verifica:
  - $A+A \cdot B=A \rightarrow$  dadas dos variables booleanas A y B  $-> A \| A \&\& B=A$
  - $A \cdot (A+B)=A \rightarrow$  dadas dos variables booleana A y B  $-> A \&\& (A \| B)=A$
- Teorema 7: para cada par de elementos de B, se verifica:
  - $A + A' \cdot B = A + B \rightarrow$  dadas dos variables booleanas A y B  $->$   

$$A \| !A \&\& B = A \| B$$
  - $A \cdot (A' + B) = A \cdot B \rightarrow$  dadas dos variables booleanas A y B  $->$   

$$A \&\& (!A \| B) = A \&\& B$$

- Teorema 8 (asociatividad): cada uno de los operadores binarios (+) y (·) cumple la propiedad asociativa:

- $A+(B+C) = (A+B)+C \rightarrow$  dadas dos variables booleanas A y B  $\rightarrow$

$$A \mid\mid (B \mid\mid C) = (A \mid\mid B) \mid\mid C$$

- $A \cdot (B \cdot C) = (A \cdot B) \cdot C \rightarrow$  dadas dos variables booleanas A y B  $\rightarrow$

$$A \&\& (B \&\& C) = (A \&\& B) \&\& C$$

Y quizá lo más importante para nosotros, porque nos permiten transformar condiciones en otras sin variar el resultado: las leyes de Demorgan:

- para cada par de elementos de B, se verifica:

- $(A+B)' = A' \cdot B' \rightarrow$  dadas dos variables booleanas A y B  $\rightarrow$

$$\neg(A \mid\mid B) = \neg A \&\& \neg B$$

- $(A \cdot B)' = A' + B' \rightarrow$  dadas dos variables booleanas A y B  $\rightarrow$

$$\neg(A \&\& B) = \neg A \mid\mid \neg B$$

## 23. Bibliografía

(Moreno, Juan Carlos 2006): Programación.

(Ceballos, Fco. Javier 2006); Java 2. Lenguaje y Aplicaciones

(Ceballos, Fco. Javier 2010): Java 2. Curso de programación

(Moya, Ricardo 2013): <https://jarroba.com/>

(Oracle API Java 8): <https://docs.oracle.com/javase/8/docs/api/>

<https://www.discoduroderoer.es>

<http://puntocomnoesunlenguaje.blogspot.com>

<https://ingenieriadesoftware.es/>

## 24. Ejercicios

- Considera el sistema que gestiona los préstamos y devoluciones de libros de una biblioteca. Cuando un usuario quiere tomar prestado un libro y no existe ningún ejemplar disponible en ese momento, el sistema permite realizar una reserva de ese libro. A medida que los libros solicitados vuelvan a estar disponibles, el sistema irá atendiendo las reservas según el orden en que se realizaron. (Solo hay que hacer el módulo de reservas)

Para ello, el sistema dispone de la clase **Reserva**, en la que se definen los siguientes atributos y métodos:

**Atributos:**

private Usuario usr, que almacena el usuario que realiza la reserva.

private int idLibro, que almacena un identificador numérico del libro reservado.

**Métodos:**

public Reserva(Usuario usr, int unIdLibro), Realiza la reserva de un libro para un usuario  
 public Usuario getUsr(), devuelve el usuario.  
 public int getIdLibro(), devuelve el atributo idLibro.  
 public boolean equals(Object obj), compara objetos de la clase.

La clase Usuario es la siguiente. Hay que terminarla. Los usuarios se tienen que poder ordenar por edad. (**No hay que implementar setter's ni getter's**)

```
public class Usuario {
    private String dni; //se da por supuesto que es único para cada usuario
    private String nombre;
    private int edad;

    Usuario(String dni, String nombre, int edad) {
        this.dni=dni;
        this.nombre=nombre;
        this.edad=edad;
    }

    public String toString(){
        return "DNI:"+dni+", Nombre: "+nombre+", Edad: "+edad;
    }

    ...
}
```

Queremos implementar una clase ReservasBiblioteca para gestionar las reservas de libros en la biblioteca. De momento, disponemos de la siguiente definición parcial:

```
public class ReservasBiblioteca {
    private Reserva[] vectorReservas; // Almacena los datos de las reservas
```

```

private int ocupados; // Indica las posiciones realmente ocupadas en el vector

public ReservasBiblioteca() {
    ocupados = 0; // Inicialmente no hay reservas
    vectorReservas = new Reserva[10]; // Crea un vector con capacidad
        // para 10 reservas
}
}

```

Como puedes ver, la representación interna de la clase ReservasBiblioteca contiene como atributos un vector de reservas (vectorReservas) y un valor entero (ocupados) que indica la cantidad de reservas en la biblioteca. De esta manera, se distingue entre la longitud (o capacidad máxima) del vector y la cantidad de reservas que realmente contiene.

ReservasBiblioteca debe de implementar la interface Ireservas que contiene los siguientes métodos:

#### **boolean anyadir(Usuario usr, int idLibro)**

Si ya existe una reserva igual en la biblioteca, devuelve como resultado false. Si no, el método añade una nueva reserva y devuelve como resultado true.

Para añadir la reserva, primero debes comprobar si el vector está lleno, en cuyo caso debes duplicar su longitud. Después, debes añadir la reserva en la primera posición libre del vector.

#### **String servirLibro(int idLibro)**

Elimina la primera reserva correspondiente al identificador de libro dado y devuelve el los datos de la persona que había realizado esa reserva. Si no hay ninguna reserva para ese identificador, el método debe devolver null.

Para eliminar una reserva del vector, todas las reservas posteriores a la misma deben desplazarse una posición a la izquierda en el mismo vector, de modo que se respete su orden de llegada.

#### **String usuarioMasReservas()**

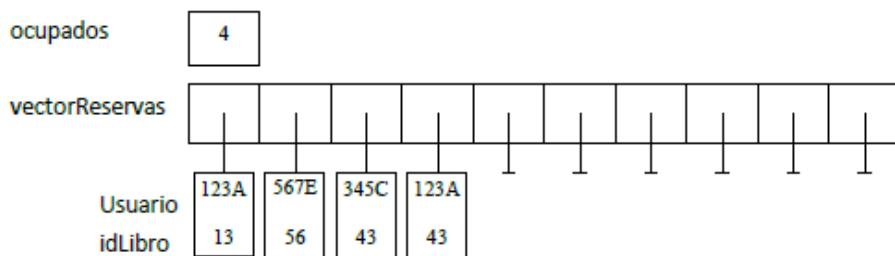
Devuelve el usuario que tiene más reservas pendientes en la biblioteca. En caso de empate entre varios usuarios, puedes devolver cualquiera de ellos. Si no existe ninguna reserva en la biblioteca, el método debe devolver como resultado null.

#### **int anularReservas(Usuario usr)**

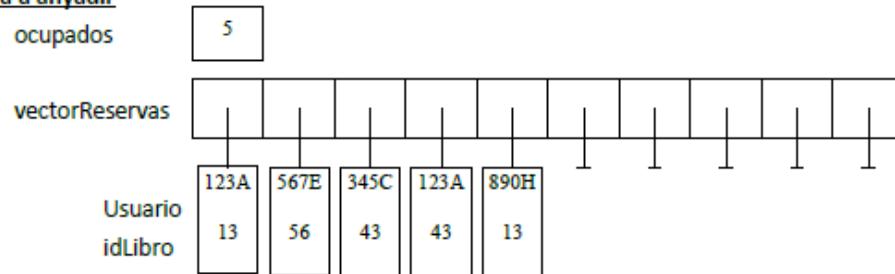
Elimina todas las reservas que existan para el usuario dado y devuelve la cantidad de reservas anuladas. Si no hay ninguna reserva para ese usuario, el método debe devolver cero. Al igual que en el método servirLibro, los desplazamientos necesarios se realizarán

sobre el propio vector (es decir, sin crear vectores auxiliares de reservas). Ejemplo:

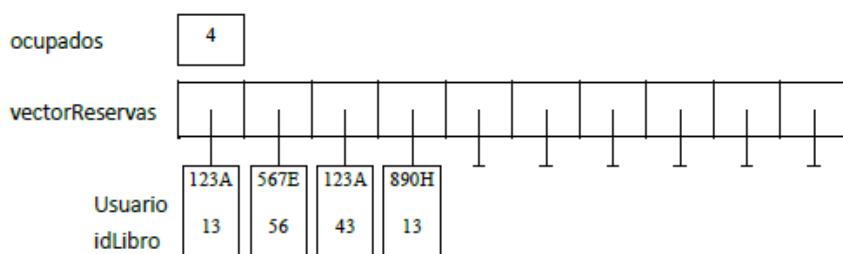
### Ejemplo



### Llamada a `anyadir`

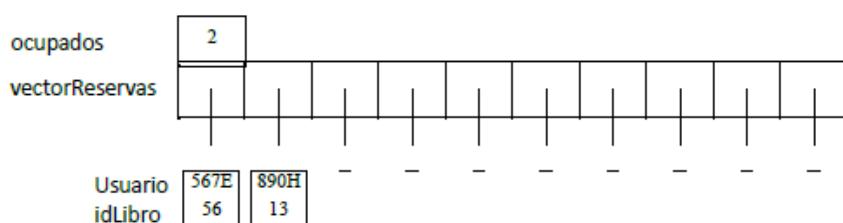


### Llamada a `servirLibro(43)`



La llamada a `usuarioMasReservas()`: devolverá--->DNI:123, Nombre: Pepe, Edad: 55

La llamada a `anularReservas("123A")`



<https://www.discoduroderoer.es/ejercicios-propuestos-y-resueltos-basicos-java/>

<http://puntocomnoesunlenguaje.blogspot.com/p/ejercicios.html>

