INFORMÁTICA Y COMUNICACIONES
Módulo Transversal

Programación

Mario Dorrego Martín

EDITORIAL SÍNTESIS

Programación

Consulte nuestra página web: www.sintesis.com
En ella encontrará el catálogo completo y comentado

NO fotocopies el libro

Queda prohibida, salvo excepción prevista en la ley, cualquier forma de reproducción, distribución, comunicación pública y transformación de esta obra sin contar con autorización de los titulares de la propiedad intelectual. La infracción de los derechos mencionados puede ser constitutiva de delito contra la propiedad intelectual (arts. 270 y sgs. Código Penal). El Centro Español de Derechos Reprográficos (www.cedro.org) vela por el respeto de los citados derechos.

Programación

Mario Dorrego Martín

EDITORIAL SÍNTESIS

Asesor editorial:
Juan Carlos Moreno Pérez

© Mario Dorrego Martín

© EDITORIAL SÍNTESIS, S. A.
Vallehermoso, 34. 28015 Madrid
Teléfono 91 593 20 98
http://www.sintesis.com

ISBN: 978-84-917183-5-2

Impreso en España - Printed in Spain

Reservados todos los derechos. Está prohibido, bajo las sanciones penales y el resarcimiento civil previstos en las leyes, reproducir, registrar o transmitir esta publicación, íntegra o parcialmente, por cualquier sistema de recuperación y por cualquier medio, sea mecánico, electrónico, magnético, electroóptico, por fotocopia o por cualquier otro, sin la autorización previa por escrito de Editorial Síntesis, S. A.

Índice

PRESENTACIÓN 13

1. INTRODUCCIÓN A LA PROGRAMACIÓN 15

Objetivos 15
Mapa conceptual 16
Glosario 16
1.1. Introducción 17
1.2. Orígenes de la programación 17
1.3. Paradigmas de programación 18
1.3.1. Programación estructurada 19
1.3.2. Programación modular 20
1.4. Pseudocódigo 26
1.4.1. Operadores, palabras reservadas y tipos de datos 26
1.4.2. Estructuras de control 27
1.5. Diagramas de flujo 31
1.5.1. Simbología 32
1.5.2. Estructuras de control 33
Resumen 36
Ejercicios propuestos 36
Supuestos prácticos 36
Actividades de autoevaluación 37

2. PROGRAMACIÓN ORIENTADA A OBJETOS 39

Objetivos 39
Mapa conceptual 40

Glosario 40
2.1. Introducción 41
2.2. Auge de la programación orientada a objetos 42
2.3. Características de la programación orientada a objetos 42
2.3.1. Clase 42
2.3.2. Objeto 45
2.3.3. Encapsulación 46
2.3.4. Abstracción 46
2.3.5. Reutilización o reusabilidad 47
2.3.6. Visibilidad 47
2.3.7. Relaciones entre clases 48
2.3.8. Polimorfismo 49
Resumen 50
Ejercicios propuestos 51
Supuestos prácticos 51
Actividades de autoevaluación 52

3. LENGUAJES DE PROGRAMACIÓN Y ENTORNOS DE DESARROLLO INTEGRADO 55

Objetivos 55
Mapa conceptual 56
Glosario 56
3.1. Introducción 58
3.2. Lenguajes de programación 58
3.3. Java 59
3.3.1. Descripción de la tecnología 59
3.3.2. Entorno de desarrollo JDK 60
3.3.3. Entorno de ejecución JRE 61
3.3.4. Variables de entorno PATH y CLASSPATH 62
3.3.5. Tipos de archivos empleados 63
3.3.6. Recolector de basura 63
3.4. Programación sin IDE 65
3.4.1. Compilar un programa 65
3.4.2. Crear un jar 66
3.4.3. Ejecutar un programa 67
3.5. Programación con IDE 68
3.5.1. Características y elementos 69
3.5.2. Alternativas 69
3.5.3. Configuración de NetBeans 70
3.5.4. Manejo del IDE 72
Resumen 76
Ejercicios propuestos 76
Supuestos prácticos 77
Actividades de autoevaluación 77

4. ELEMENTOS DE UN PROGRAMA INFORMÁTICO 79

Objetivos 79
Mapa conceptual 80
Glosario 80

4.1. Soluciones y proyectos ... 81
4.2. Identificadores ... 82
4.3. Palabras reservadas ... 82
4.4. Clases ... 83
4.4.1. Método *main* ... 83
4.5. Paquetes ... 84
4.6. Variables ... 85
4.6.1. Ámbito de una variable ... 85
4.7. Constantes ... 86
4.8. Tipos de datos primitivos ... 86
4.8.1. Conversiones de tipo (*casting*) ... 87
4.9. *String* ... 88
4.10. Clases envoltorio ... 89
4.10.1. Autoboxing - Unboxing ... 90
4.11. Secuencias de escape ... 90
4.12. Comentarios ... 90
4.13. Operadores y expresiones ... 91
4.13.1. Precedencia de operadores ... 92
Resumen ... 93
Ejercicios propuestos ... 93
Supuestos prácticos ... 94
Actividades de autoevaluación ... 94

5. ESTRUCTURAS DE CONTROL ... 97

Objetivos ... 97
Mapa conceptual ... 98
Glosario ... 98
5.1. Introducción ... 99
5.2. Entrada y salida de información ... 99
5.2.1. Entrada por argumentos ... 99
5.2.2. Entrada y salida por consola ... 101
5.3. Estructuras de selección (alternativa) ... 104
5.3.1. *if, else, else if* ... 104
5.3.2. *switch* ... 105
5.4. Estructuras de repetición (iterativa) ... 106
5.4.1. *for* ... 106
5.4.2. *while* ... 106
5.4.3. *do while* ... 107
5.5. Estructuras de salto incondicional ... 107
5.5.1. *break continue* ... 107
5.5.2. *break continue* con etiquetas ... 108
5.6. Prueba y depuración de programas ... 109
Resumen ... 110
Ejercicios propuestos ... 110
Supuestos prácticos ... 110
Actividades de autoevaluación ... 111

6. CLASES Y UTILIZACIÓN DE OBJETOS ... 113

Objetivos ... 113
Mapa conceptual ... 114
Glosario ... 114
6.1. Concepto de clase ... 115
6.2. Estructura y miembros de una clase ... 115
6.2.1. Campos ... 116
6.2.2. Métodos ... 117
6.2.3. Constructores e instanciación de objetos ... 120
6.2.4. Destructores y liberador de memoria ... 121
6.2.5. Propiedades ... 122
6.3. Encapsulación y visibilidad ... 123
6.4. Librerías y paquetes de clases ... 124
Resumen ... 125
Ejercicios propuestos ... 125
Supuestos prácticos ... 125
Actividades de autoevaluación ... 126

7. CLASES AVANZADAS Y UTILIZACIÓN DE OBJETOS ... 129

Objetivos ... 129
Mapa conceptual ... 130
Glosario ... 130
7.1. Asociación, agregación y composición de clases ... 131
7.2. Herencia ... 133
7.3. Constructores y herencia ... 133
7.4. Acceso a campos de la superclase ... 135
7.5. Acceso a métodos de la superclase ... 135
7.6. Sobreescritura de métodos (*Override*) ... 136
7.7. Clases y métodos abstractos ... 136
7.7.1. Métodos abstractos ... 137
7.8. Clases y métodos finales ... 137
7.8.1. Métodos finales ... 137
7.9. Interfaces ... 138
7.9.1. Pseudoherencia múltiple ... 140
7.10. Polimorfismo ... 141
7.10.1. Asignación polimorfa ... 141
7.10.2. Ejecución polimorfa ... 141
Resumen ... 142
Ejercicios propuestos ... 143
Supuestos prácticos ... 143
Actividades de autoevaluación ... 144

8. ESTRUCTURAS DE ALMACENAMIENTO: *ARRAYS* Y CADENAS DE CARACTERES ... 147

Objetivos ... 147
Mapa conceptual ... 148
Glosario ... 148
8.1. Introducción ... 149

8.2. Estructuras ... 149
8.3. *Arrays* ... 150
8.3.1. Dimensiones de un *array* ... 150
8.3.2. Declaración y creación ... 150
8.3.3. Inicialización y acceso ... 151
8.3.4. *Arrays* y métodos ... 152
8.3.5. Operaciones para realizar sobre un *array* estático ... 152
8.4. Cadenas de caracteres ... 159
8.4.1. Declaración, creación e inicialización ... 159
8.4.2. Operaciones ... 159
8.4.3. Conversiones ... 162
Resumen ... 163
Ejercicios propuestos ... 163
Supuestos prácticos ... 163
Actividades de autoevaluación ... 164

9. COLECCIONES Y TIPOS ABSTRACTOS DE DATOS ... 167

Objetivos ... 167
Mapa conceptual ... 168
Glosario ... 168
9.1. Colecciones y tipos abstractos de datos ... 169
9.2. Interfaz *Collection* ... 170
9.2.1. Interfaz *Set* ... 171
9.2.2. Interfaz *List* ... 173
9.2.3. Interfaz *Queue* ... 180
9.3. Interfaz *Map* ... 181
9.3.1. *HashMap* ... 182
9.3.2. *LinkedHashMap* ... 182
9.3.3. *TreeMap* ... 183
Resumen ... 184
Ejercicios propuestos ... 185
Supuestos prácticos ... 185
Actividades de autoevaluación ... 186

10. CONTROL Y MANEJO DE EXCEPCIONES ... 189

Objetivos ... 189
Mapa conceptual ... 190
Glosario ... 190
10.1. Excepciones ... 191
10.2. Jerarquía de excepciones ... 191
10.3. Miembros de una excepción ... 193
10.4. Manejo de excepciones ... 193
10.4.1. Captura de excepciones ... 193
10.4.2. Propagación de excepciones ... 196
10.4.3. Lanzamiento de excepciones ... 197
10.4.4. Creación de clases de excepciones ... 198
10.5. Recomendaciones de uso de excepciones ... 199
Resumen ... 199

Ejercicios propuestos ... 200
Supuestos prácticos ... 201
Actividades de autoevaluación ... 201

11. RECURSIVIDAD Y COMPLEJIDAD ALGORÍTMICA ... 203

Objetivos ... 203
Mapa conceptual ... 204
Glosario ... 204
11.1. Introducción ... 205
11.2. Concepto de recursividad ... 205
11.3. Utilización de la recursividad ... 206
11.4. Tipos de recursividad ... 207
11.5. Ventajas e inconvenientes ... 209
11.6. Complejidad algorítmica ... 210
Resumen ... 212
Ejercicios propuestos ... 213
Supuestos prácticos ... 213
Actividades de autoevaluación ... 214

12. LECTURA Y ESCRITURA DE INFORMACIÓN ... 217

Objetivos ... 217
Mapa conceptual ... 218
Glosario ... 218
12.1. Introducción ... 219
12.2. Flujos de comunicación ... 219
12.2.1. Tipos de flujo ... 219
12.2.2. Flujos predeterminados ... 220
12.2.3. Utilización de flujos ... 221
12.2.4. Clases relativas a flujos ... 221
12.3. Aplicaciones del almacenamiento de información en ficheros ... 229
12.3.1. Tipos de ficheros. Registro ... 230
12.3.2. Creación y eliminación de ficheros y directorios ... 230
12.3.3. Apertura y cierre de ficheros. Modos de acceso ... 232
12.3.4. Escritura y lectura de información en ficheros ... 235
12.3.5. Almacenamiento de objetos en ficheros. Serialización ... 238
12.3.6. Utilización de los sistemas de ficheros ... 240
Resumen ... 241
Ejercicios propuestos ... 241
Supuestos prácticos ... 242
Actividades de autoevaluación ... 243

13. MANTENIMIENTO DE LA PERSISTENCIA DE LOS OBJETOS ... 245

Objetivos ... 245
Mapa conceptual ... 246
Glosario ... 246
13.1. Introducción ... 247
13.2. Base de datos orientada a objetos ... 247

13.3. Características de las bases de datos orientadas a objetos ... 248
13.4. Instalación del gestor de bases de datos ... 248
13.5. Creación de bases de datos ... 249
13.6. El lenguaje de definición de objetos ... 250
13.6.1. Tipos de datos ... 250
13.6.2. Clases ... 251
13.6.3. Campos ... 253
13.6.4. Constructores ... 256
13.6.5. Herencia ... 256
13.7. Mecanismos de consulta ... 256
13.7.1. El lenguaje de consultas: sintaxis, expresiones, operadores ... 257
13.7.2. Establecimiento de la conexión ... 262
13.7.3. Inserción, recuperación, modificación y borrado de información ... 263
13.8. Explorador de objetos ... 268
Resumen ... 269
Ejercicios propuestos ... 269
Supuestos prácticos ... 269
Actividades de autoevaluación ... 270

14. GESTIÓN DE BASES DE DATOS RELACIONALES ... 273

Objetivos ... 273
Mapa conceptual ... 274
Glosario ... 274
14.1. Introducción ... 275
14.2. Base de datos relacional ... 275
14.3. Creación de bases de datos ... 276
14.4. Establecimiento de la conexión ... 277
14.5. Ejecución de consultas sobre la base de datos ... 279
14.6. Mecanismos de actualización de la base de datos ... 280
14.7. Recuperación de información ... 282
14.8. Manipulación de la información ... 283
14.9. Utilización de asistentes ... 283
Resumen ... 285
Ejercicios propuestos ... 285
Supuestos prácticos ... 286
Actividades de autoevaluación ... 286

15. CREACIÓN DE INTERFACES GRÁFICAS ... 289

Objetivos ... 289
Mapa conceptual ... 290
Glosario ... 290
15.1. Introducción ... 291
15.2. Librería *AWT* ... 291
15.2.1. Ventanas, contenedores y controles ... 292
15.2.2. Eventos ... 294
15.2.3. *Layouts* ... 297
15.2.4. Menús ... 299
15.3. Librería *Swing* ... 301
15.3.1. Ventanas, contenedores y controles ... 301
15.3.2. Eventos ... 304
15.3.3. Layouts ... 304
15.3.4. Modificando la apariencia ... 310
15.4. Herramientas gráficas ... 311
Resumen ... 313
Ejercicios propuestos ... 313
Supuestos prácticos ... 314
Actividades de autoevaluación ... 315

Presentación

Atrás han quedado muchas de las técnicas y procedimientos a seguir para la resolución de determinados problemas. Hoy en día, gracias al rápido desarrollo de la tecnología y a la cada vez más creciente demanda de automatización de procesos, la programación se ha posicionado como una de las actividades más demandadas.

Este libro ha sido creado con el objetivo de permitir al alumno aprender a desarrollar programas. Gracias a él podrá afrontar con garantías de éxito el Módulo Profesional *Programación* (0485), contenido en los ciclos formativos de grado superior *Técnico Superior en Desarrollo de Aplicaciones Web* y *Técnico Superior en Desarrollo de Aplicaciones Multiplataforma*, cuyos títulos pueden consultarse respectivamente en el Real Decreto 686/2010 y en el Real Decreto 450/2010.

Dada la temática tratada, los contenidos y los ejemplos incluidos, este libro podrá ser utilizado también por alumnos de los certificados de profesionalidad *Programación en lenguajes estructurados de aplicaciones de gestión* (IFCD0111) y *Programación con lenguajes orientados a objetos y bases de datos relacionales* (IFCD0112). Asimismo, podrá ser utilizado como material de estudio por estudiantes y profesionales que necesiten preparar su examen de certificación Oracle, así como para cualquier estudiante que necesite aprender a desarrollar aplicaciones en cualquiera de las enseñanzas regladas y no regladas. Incluso si ya tiene experiencia en el desarrollo de aplicaciones, los temas tratados pueden ofrecerle una visión actualizada de las tecnologías que esté usando o servirle de guía con la que dar el salto a Java desde otras tecnologías (C++, C#, Delphi…).

Los contenidos han sido estructurados en quince capítulos. A lo largo de los dos primeros se introducirán los diferentes paradigmas de programación, haciéndose gran hincapié en el paradigma de programación orientada a objetos. Como técnicas introductorias con las que tener una primera toma de contacto en el desarrollo de algoritmos y para la obtención de soluciones independientes de la tecnología final, se han utilizado las técnicas del pseudocódigo y los flujogramas.

Aunque son muchas las tecnologías que permiten al alumno encarar el reto de aprender a programar, se ha decidido utilizar Java como lenguaje vehicular con el que comenzar a desarrollar programas reales y materializar los diagramas y pseudoprogramas. Su elección no ha

sido algo dejado al azar; se ha apostado por una tecnología ampliamente implantada, con gran proyección de futuro y con la que sin duda alguna, su valía profesional se verá totalmente avalada. Será en el capítulo 3 donde se abordará de lleno la tecnología Java, exponiéndose sus características principales, así como las herramientas disponibles para el desarrollo, compilación y ejecución de código.

Ya en los capítulos 4 y 5 se describirán los tipos empleados por el lenguaje, los operadores, los comentarios, las diferentes estructuras de control y, en líneas generales, la sintaxis empleada por el lenguaje.

Aunque la programación orientada a objetos es tratada de forma genérica en el capítulo 2, no será hasta los capítulos 6 y 7 cuando se muestre cómo implementar este paradigma sobre la tecnología elegida.

Los capítulos 8 y 9 tienen como objetivo principal mostrar las estructuras de almacenamiento, estáticas y dinámicas, empleadas en el manejo de información de un programa. En ellos serán descritas y analizadas en detalle las colecciones con las que conseguir implementar los diferentes tipos abstractos de datos (pila, cola, lista…).

Cualquier programa es susceptible de verse afectado por eventos o situaciones imprevistas que pueden originar una salida anómala del mismo. En el capítulo 10 se detallarán los mecanismos disponibles para controlar este tipo de comportamientos.

Una de las características más potentes de los lenguajes de programación como Java es la posibilidad que estos ofrecen para resolver determinados problemas basándose en llamadas recurrentes del mismo algoritmo. En el capítulo 11 se estudiará a fondo el concepto de *recursividad*, así como el de *complejidad algorítmica* como medida de la calidad de un algoritmo.

La inmensa mayoría de los programas reales requiere que la información manejada sea almacenada en algún soporte desde el cual pueda recuperarse posteriormente. En los capítulos 12, 13 y 14 se detallará cómo manejar la información contenida en diferentes fuentes (ficheros de texto, ficheros binarios, bases de datos orientadas a objetos y bases de datos relacionales).

Será en el último de los capítulos donde se introducirá al lector en el desarrollo de interfaces gráficas de usuario con las que hacer más cómodo y atractivo el manejo de los programas.

Además, en la página web de la editorial (www.sintesis.com) se incluyen varios recursos adicionales. Estos contenidos están especificados en recuadros llamados *Recurso digital* dentro del capítulo y pueden consultarse siguiendo las indicaciones de la primera página del libro.

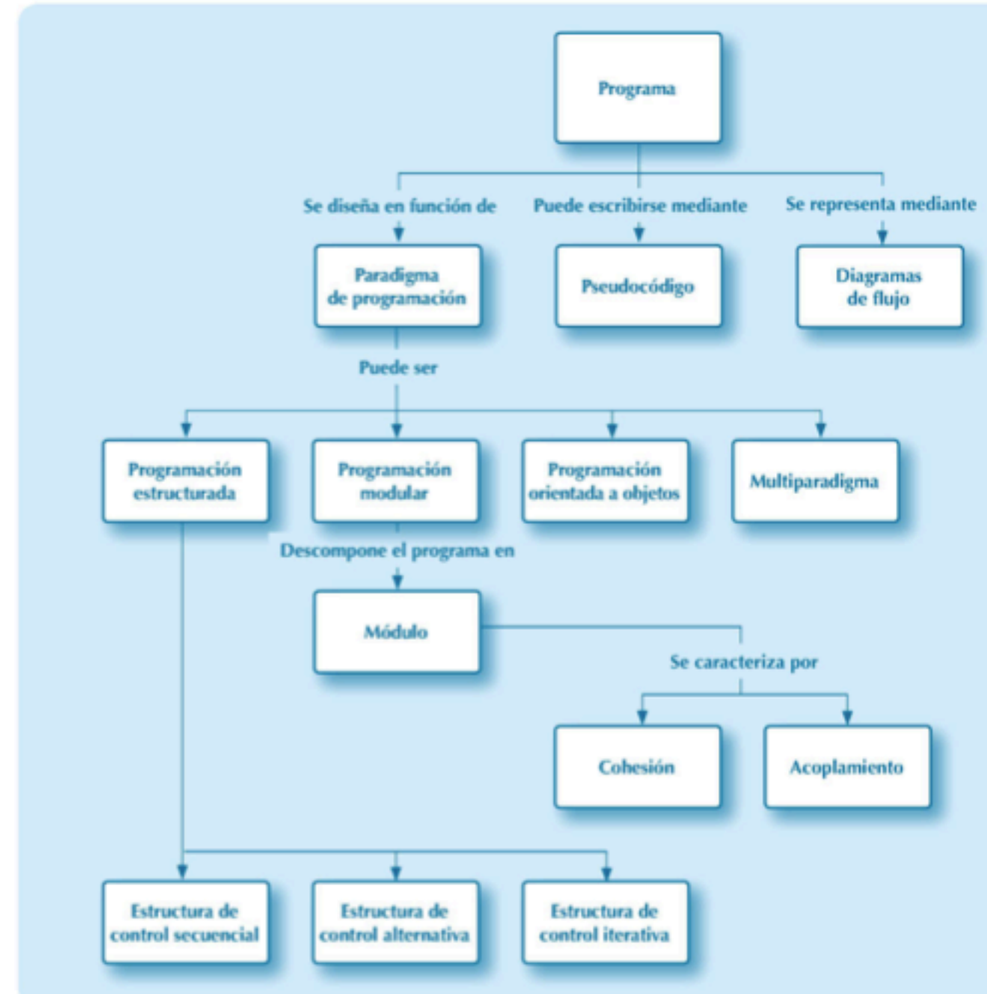
1

Introducción a la programación

Objetivos

- ✓ Conocer los conceptos básicos relacionados con la programación y el diseño de aplicaciones.
- ✓ Describir los paradigmas de programación más utilizados.
- ✓ Aprender a utilizar sistemas de descripción de programas de alto nivel.

Mapa conceptual



Glosario

Acoplamiento. Grado y forma de dependencia entre los módulos que comparten información.

ANSI. Instituto Nacional Americano de Estándares. Es la organización encargada del desarrollo de estándares para productos, servicios y procesos.

Argumento. Valor transferido a un parámetro correspondiente a un procedimiento o función cuando este es invocado.

Aritmómetro. Evolución de la antigua calculadora mecánica de Leibniz, capaz de realizar las cuatro operaciones aritméticas elementales (suma, resta, multiplicación y división) con resultados de hasta 12 cifras.

Booleano. Es el tipo que puede tomar una variable o condición. Se limita a 2 valores: verdadero o falso.

Código espagueti. Código de un programa cuya estructura es tan compleja que hace muy difícil su seguimiento y modificación.

Cohesión. Nivel de relación existente entre los elementos software contenidos en un módulo.

Diagrama. Representación gráfica de un conjunto de elementos y las relaciones existentes entre sí.

Máquina analítica. Computador moderno de naturaleza mecánica ideado por Charles Babbage, que dispone de la mayor parte de los elementos lógicos de un ordenador tal como hoy en día se conoce.

Paradigma. Guía de trabajo a seguir para la obtención de un resultado, elaborada y consensuada por un grupo de expertos en la materia.

Parámetro. Propiedad de un procedimiento o función que se define junto a él y que establece la forma y el tipo del valor que espera recibir cuando es invocado.

Programación. Proceso de diseño y codificación de las instrucciones que se ejecutan sobre un computador.

Pseudocódigo. Lenguaje cercano a un lenguaje de programación cuyo objetivo es el desarrollo de algoritmos fácilmente interpretables por un programador.

Teorema. Término proveniente del latín *theorēma* que referencia a una verdad que, sin ser obvia, es totalmente demostrable.

1.1. Introducción

En este primer capítulo se realizará una breve introducción a la programación desde sus orígenes hasta los tiempos modernos, se presentarán conceptos generales de programación y se abordarán los diferentes paradigmas de programación, explicando en detalle la programación estructurada y la programación modular. Por último, se expondrán los diagramas de flujo y el pseudocódigo como técnicas que ayudarán a introducirse en el mundo de la programación.

1.2. Orígenes de la programación

Se hace uso del término *programación* para hacer referencia al proceso mediante el cual se diseña y codifica un conjunto de instrucciones que implementan un determinado algoritmo.

Surgió ante la necesidad de automatizar y facilitar las tareas que inicialmente se realizaban de forma manual. La programación tiene sus raíces en las primeras calculadoras de operaciones matemáticas elementales, las cuales llevaban los algoritmos implementados en su estructura física, y en los telares automáticos, que recibían los datos sobre el trabajo a realizar a través de tarjetas perforadas. Sin embargo, no es hasta la aparición del diseño de la *máquina analítica* de Charles Babbage (considerado el primer ordenador de la historia) y la posterior aportación de Augusta Ada Byron (considerada la primera programadora de la historia) cuando puede hablarse de programación como tal.

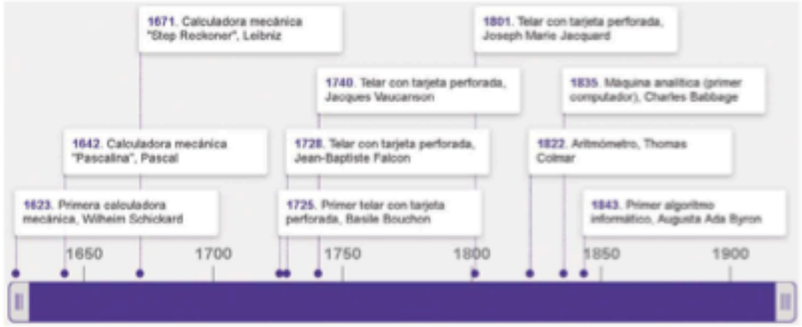


Figura 1.1
Orígenes de la programación.

Hoy en día el concepto de *programación* es sustancialmente diferente, y ello se debe a que ha ido evolucionando de forma pareja a como lo han ido haciendo los dispositivos y tecnologías de cada época (figura 1.1). En las primeras computadoras las instrucciones de los programas eran escritas en código binario (ceros y unos). Dada la complicación asociada a esta técnica de programación, los científicos que estudiaban esta nueva ciencia decidieron sustituir determinadas secuencias binarias por palabras que permitieran representarlas más fácilmente; este proceso dio lugar al conocido *lenguaje ensamblador*. Conforme las necesidades fueron creciendo y se fue depurando la técnica aparecieron los lenguajes de alto nivel, que añadían una nueva capa de abstracción al proceso de programación y que permitían el desarrollo de programas cada vez más sofisticados al mismo tiempo que más fáciles de desarrollar.

1.3. Paradigmas de programación

A lo largo de la historia de la computación se han empleado diversas técnicas de programación, presentando cada una de ellas unas características concretas que las hace diferenciarse del resto y mostrarse como candidatas ideales para según qué problema. Cuando un conjunto de reglas y costumbres son aceptadas como modelo de referencia para generar un código de calidad, se puede comenzar a hablar de un *paradigma de programación*.

Pero, a pesar de que cada paradigma de programación tiene unas características concretas, en la vida real los programadores pueden basarse en varios de ellos al mismo tiempo para el desarrollo de sus programas. La inclusión de dos o más paradigmas de programación en el mismo programa se denomina *programación multiparadigma* y generalmente es la que mejores resultados obtiene.

Utilizar un paradigma de programación, aceptando y siguiendo sus reglas, derivará en un código fuente fácil de mantener, entender y corregir.

Son muchos los paradigmas de programación que se pueden enumerar, aunque a continuación serán descritos con detalle aquellos más empleados y que mayor relación guardan con los contenidos del curso. En este capítulo se estudiará en detalle el *paradigma de programación estructurada* y el *paradigma de programación modular*. El *paradigma de programación orientada a objetos* será estudiado en el capítulo 2.

Actividad propuesta 1.1

Busca ejemplos de otros paradigmas de programación diferentes a los mencionados anteriormente y establece una línea de tiempo que muestre el momento en el que fueron surgiendo.

1.3.1. Programación estructurada

La *programación estructurada* tiene sus orígenes en la década de los sesenta y pretende mejorar la calidad del software, así como agilizar su desarrollo a la vez que reduce su coste. Deriva del *paradigma de programación imperativa*, en el cual se especifica cómo debe resolverse el problema mediante un conjunto de instrucciones que son ejecutadas y que van variando el estado del programa.

La programación estructurada se basa en el *teorema del programa estructurado* propuesto por Corrado Böhm y Giuseppe Jacopini, según el cual todo programa se puede escribir empleando únicamente tres tipos básicos de estructuras de control:

1. *Secuencial*: las instrucciones se ejecutan una detrás de otra siguiendo un orden (secuencialidad).
2. *Alternativa:* las expresiones son evaluadas y, dependiendo del resultado, se decide cuál será la siguiente instrucción a ejecutar.
3. *Iterativa:* se repetirá un conjunto de instrucciones hasta que una condición sea cierta, permitiéndose de esta manera el salto a otra instrucción.

Aunque los lenguajes de programación ofrecen un abanico más amplio de estructuras de control, bien es cierto que todas ellas finalmente se basan en las anteriormente descritas.

Investiga

Edsger Dijkstra escribió "La sentencia *Go To* considerada dañina". Busca información acerca de ese escrito y extrae las ideas principales.

Los lenguajes de programación también ofrecen instrucciones de *salto incondicional* (*go to*), pero su uso está totalmente desaconsejado porque complica el seguimiento y trazabilidad de los programas, generando un *código espagueti* más difícil de modificar y corregir. Es por ello por lo que este tipo de estructuras deben evitarse en la medida de lo posible.

Por último, hay que reseñar que la programación estructurada presenta grandes dificultades a la hora de abordar proyectos de gran tamaño, para los cuales es más eficiente el empleo de la programación orientada a objetos.

Recuerda

✓ En la programación estructurada los datos y las funciones que los manejan se definen separados.

La programación estructurada no ofrece garantías para abordar proyectos de gran tamaño, en cuyo caso es más recomendable emplear programación orientada a objetos.

1.3.2. Programación modular

La programación estructurada establece unas directrices base para realizar un código de calidad y facilitar el mantenimiento. Aun así, no tiene en cuenta que el código puede crecer hasta llegar a complicar mucho el desarrollo.

La *programación modular* propone como solución la descomposición del problema en subproblemas de menor tamaño, los cuales serán descompuestos sucesivamente en otros de menor tamaño hasta que el resultado de dichas descomposiciones permita obtener problemas fáciles de resolver. A esta técnica de resolución de problemas se le denomina *divide y vencerás* y permite poner en práctica un proceso de desarrollo software *top-down*.

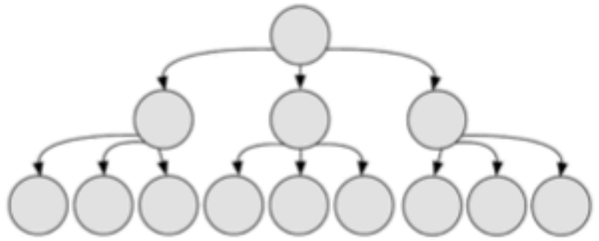


Figura 1.2
Top-down.

Cada bloque de código que se ocupará de resolver un problema de menor tamaño se denomina *módulo* o *subprograma*, y estará compuesto por un conjunto de sentencias físicamente unidas y delimitadas. Dicho módulo será referenciado por un nombre y podrá hacer uso de otros módulos. La comunicación entre módulos se realizará a través de interfaces de comunicación claramente definidas.

Los elementos empleados para llevar a cabo la modularización pueden ser muy distintos dependiendo del nivel de abstracción:

1. A un alto nivel serán las *unidades*, en diseños estructurados, y los *paquetes* y *librerías*, en diseños orientados a objetos.
2. A más bajo nivel serán los *procedimientos* y las *funciones*, en diseños estructurados, y las *clases* con sus correspondientes *métodos*, en diseños orientados a objetos.

Para llevar a cabo una correcta modularización del programa, es necesario realizar un diseño de calidad que tenga en cuenta los siguientes criterios:

- El módulo debe tener un único punto de entrada y un único punto de salida.
- Cada módulo debe realizar una única función bien definida.
- Cada módulo debe comportarse como una caja negra, de manera que dependa únicamente de las entradas.
- Módulos trazables en una pantalla (tamaño ideal 30-50 líneas de código por módulo).
- Máxima *cohesión* y mínimo *acoplamiento*. Ambos criterios están relacionados entre sí de manera que, a mayor cohesión en los módulos, menor será el acoplamiento entre ellos.

A) Cohesión

Se denomina *cohesión* de un módulo al nivel de relación existente entre los elementos software contenidos en él, entendiéndose como elementos software las instrucciones, definiciones de datos o las llamadas a otros módulos.

Existen diferentes tipos de cohesión:

1. *Cohesión funcional:* los elementos del módulo contribuyen a la realización de una única tarea. Dicho de otra manera, cada elemento es una parte integral de la estructura del módulo. El módulo se representa mediante un identificador simple. Por ejemplo: módulos matemáticos *suma, seno, potencia*…
2. *Cohesión secuencial:* se da cuando el módulo realiza varias tareas según una secuencia que establece que la salida de una actividad es la entrada necesaria de la siguiente. Por ejemplo:

Progresivamente el módulo se ocupará de buscar información a partir de un dato de entrada y, con ella, generar un informe formateado que devolverá como dato de salida. Podrían encadenarse más acciones.

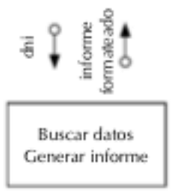


Figura 1.3
Cohesión secuencial.

3. *Cohesión comunicacional:* contiene actividades paralelas (sin orden) que comparten los mismos datos (de entrada o salida). Se recomienda su descomposición en módulos independientes de cohesión funcional. Por ejemplo:

El módulo se ocupará de buscar información a partir de un dato de entrada y retornará como salida información de diverso tipo. El orden de las actividades no es relevante.

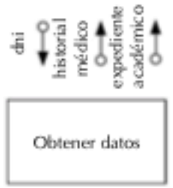


Figura 1.4
Cohesión comunicacional.

4. *Cohesión procedural:* los elementos realizan diferentes actividades posiblemente no relacionadas entre sí. También es posible que no exista relación alguna entre los datos de entrada y de la salida de los módulos. El control fluye de una actividad a la siguiente. Por ejemplo:

El módulo eliminará del sistema los datos relativos al libro cuyo código recibe como dato de entrada; posteriormente retornará como datos de salida un listado de los socios de la biblioteca.

Figura 1.5
Cohesión procedural.

5. *Cohesión temporal:* los elementos están implicados en la realización de actividades relacionadas por el momento en el cual se llevan a cabo. Ejemplos típicos son los módulos de inicialización y finalización de sistema (arrancar sistema, apagar sistema, resetear sistema, inicializar sistema…). Es posible que no manejen ningún dato de entrada o salida.

6. *Cohesión lógica:* los elementos están destinados a que realicen actividades de una misma categoría general, pero la selección de la actividad concreta tiene lugar desde fuera del módulo. Por ejemplo:

El módulo se encargará de efectuar un pago que podrá realizarse en efectivo o por transferencia (el medio de pago se seleccionará desde fuera del módulo).

Figura 1.6
Cohesión lógica.

7. *Cohesión casual:* los elementos no guardan ninguna relación observable y son fruto de una organización caótica. Por ejemplo:

Dentro del módulo se realizarán tareas de diverso tipo pero sin relación alguna. La selección de la tarea en cuestión se realiza mediante un *flag* creado para tal efecto. Algunas tareas pueden devolver datos mientras que otras no.

Figura 1.7
Cohesión casual.

Aunque siempre se tiende a pensar que, a mayor cohesión, mejor será el diseño de un programa, hay que tener en cuenta que no todos los tipos de cohesión son deseables. En la siguiente escala se pueden apreciar los diferentes tipos de cohesión y cómo estos afectan al mantenimiento.

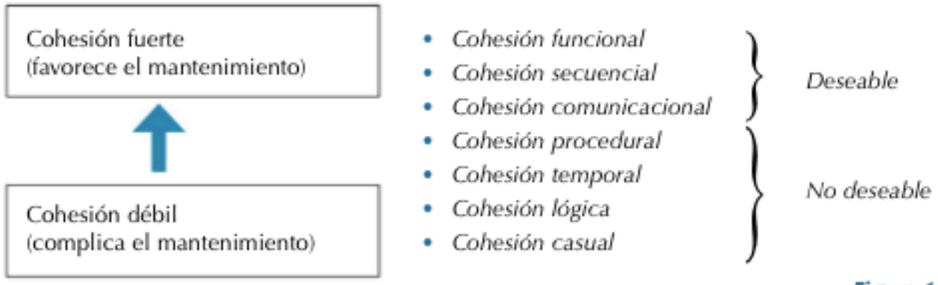


Figura 1.8
Escala de cohesión por Stevens, Myers, Constantine y Yourdon.

Mediante el siguiente árbol de cohesión se podrá identificar el tipo de cohesión ante el que un programador puede encontrarse:

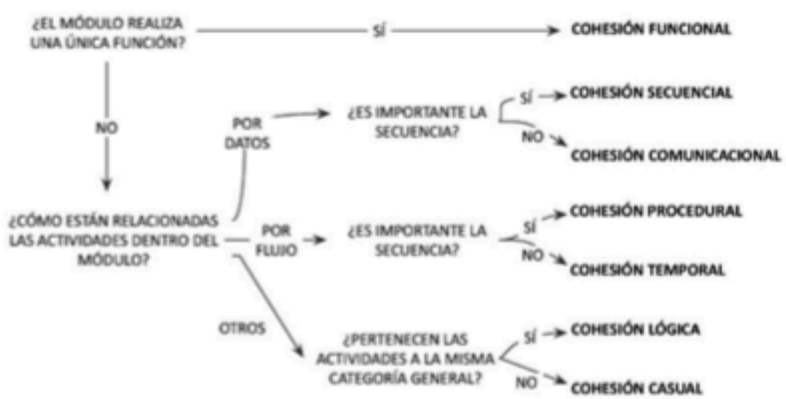


Figura 1.9
Árbol de cohesión.

Actividad propuesta 1.2

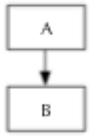
Propón dos ejemplos de cada uno de los tipos de cohesión vistos anteriormente.

B) Acoplamiento

El término *acoplamiento* hace referencia al grado y forma de dependencia entre los módulos. A menor cantidad de información compartida entre diferentes módulos, menor acoplamiento y por tanto mejor será el diseño.

Existen varios tipos de acoplamiento que son descritos a continuación:

1. *Acoplamiento normal:* se da en aquellos casos en los que un módulo *A* invoca a otro módulo *B*. Por ejemplo:



El módulo *A* invoca al módulo *B*, el cual, tras realizar su función, retorna el control al módulo *A*.

Figura 1.10
Acoplamiento normal.

De intercambiarse información, esta únicamente estará presente en los parámetros de llamada. En ese caso, y atendiendo al tipo de la información, se definen tres subtipos:

a) *Acoplamiento normal por datos:* si los parámetros intercambiados son datos elementales (tipos básicos). Por ejemplo:

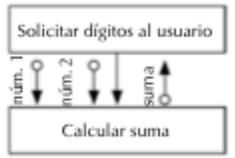


Figura 1.11
Acoplamiento normal por datos.

El módulo "Solicitar dígitos al usuario" pide datos simples al usuario (números en este caso) y los almacena. Posteriormente invoca al módulo "Calcular suma" pasándole como parámetros los datos solicitados.

El módulo "Calcular suma" hace la operación indicada y devuelve el resultado.

b) *Acoplamiento normal por marca o estampado:* si los parámetros intercambiados son un dato compuesto (registro) de datos de tipos básicos. Por ejemplo:

Figura 1.12
Acoplamiento normal por marca o estampado.

El módulo "Pedir datos al usuario" solicita datos simples al usuario y los almacena en forma de dato compuesto (registro). Posteriormente invoca al módulo "Insertar datos en BD" pasándole como parámetro el dato compuesto.

El módulo "Insertar datos en BD" inserta los datos del registro en la base de datos y retorna el identificador asignado a la inserción realizada.

c) *Acoplamiento normal de control:* cuando un módulo le pasa al otro un parámetro con la intención de controlar su lógica de funcionamiento interna. Por ejemplo:

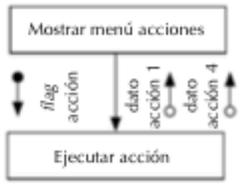


Figura 1.13
Acoplamiento normal de control.

El módulo "Mostrar menú acciones" presenta las acciones que se pueden realizar y recoge la que el usuario seleccione. Posteriormente invoca al módulo "Ejecutar acción" pasándole como parámetro el identificador de la acción que el usuario seleccionó previamente.

El módulo "Ejecutar acción" lleva a cabo la acción correspondiente y, dependiendo de esta, puede que retorne algún dato.

2. *Acoplamiento externo*: se da cuando dos o más módulos utilizan las mismas fuentes externas de datos (interfaces de dispositivos o de programas externos).
3. *Acoplamiento global*: presente cuando los módulos utilizan los mismos datos globales (variables globales, memoria compartida, ficheros o bases de datos).
4. *Acoplamiento patológico o por contenido*: un módulo lee o modifica los datos internos de otro módulo o bien salta al interior de su código.

En la figura 1.14 se pueden apreciar los diferentes tipos de acoplamiento y cómo estos afectan al mantenimiento:

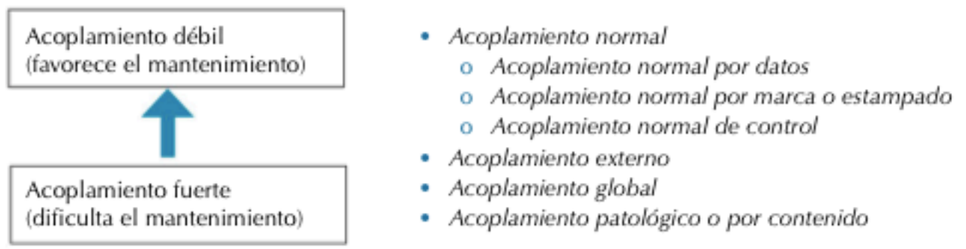


Figura 1.14
Tipos de acoplamiento.

Recuerda

✓ La programación modular intenta solventar el crecimiento desmesurado que se plantea con la programación estructurada mediante la *descomposición* en módulos o subprogramas de menor tamaño.

Cuanto mayor sea la relación existente entre los elementos contenidos en un módulo, mayor será la *cohesión* y, por tanto, más exitosa la modularización llevada a cabo.

Existen varios tipos de cohesión. Steven, Myers, Constantine y Yourdon (1974) establecieron la "escala de cohesión", que identifica qué tipos son los más deseables así como aquellos que deben evitarse.

Mediante el *árbol de cohesión* es fácil conocer el tipo de cohesión.

Cuanto menor sea la relación existente entre los elementos de un módulo con los elementos de otro módulo, menor será el *acoplamiento* y, por tanto, más exitosa la modularización realizada.

Existen varios tipos de acoplamiento y los módulos pueden estar relacionados por varios tipos de ellos, en cuyo caso el acoplamiento que caracterizará la relación entre estos será la del peor tipo.

Actividad propuesta 1.3

Propón dos ejemplos de cada uno de los tipos de acoplamiento vistos anteriormente.

1.4. Pseudocódigo

El *pseudocódigo* es un lenguaje cercano a un lenguaje de programación cuyo objetivo es el desarrollo de algoritmos fácilmente interpretables por un programador, independientemente del lenguaje de programación del que provenga. En sí mismo no se trata de un lenguaje de programación, pero sí que utiliza un conjunto limitado de expresiones que permiten representar las estructuras de control y los módulos descritos en los paradigmas de programación estructurada y modular.

Mediante pseudocódigo se puede escribir aquellos algoritmos que tengan solución finita y que comiencen desde un único punto de partida. La escritura de un algoritmo o programa en pseudocódigo debería favorecer la posterior traducción al lenguaje de programación elegido.

Sabías que...

Dado que no existe una sintaxis estandarizada para la escritura de pseudocódigo, es posible encontrar diferencias sustanciales en los pseudocódigos escritos por diferentes programadores.

1.4.1. Operadores, palabras reservadas y tipos de datos

A pesar de que no existe una norma rígida que establezca cómo realizar la escritura de programas en pseudocódigo, es recomendable seguir una serie de recomendaciones que permitan transcribir el programa al lenguaje de programación que va a usarse con la mayor facilidad. A continuación, se muestran las tablas resumen de los operadores, palabras reservadas y tipos de datos empleados para la escritura de pseudocódigo en el capítulo.

Cuadro 1.1
Resumen operadores

Aritméticos		Relacionales (Usados para formar condiciones)		Lógicos (Usados para formar condiciones)	
+	Suma				
-	Resta	=	Igual	and	y lógico (conjunción)
*	Multiplicación	<	Menor	or	o lógico (disyunción)
/	División real	≤	Menor o igual	no	Negación lógica
div	División entera	>	Mayor	**Especiales**	
mod o %	Resto o módulo	≥	Mayor o igual	←	Asignación
^	Potencia	<>	Distinto	//	Comentario

Cuadro 1.2
Palabras reservadas

Inicio	Si no	Otro	Para	En
Fin	Según	Mientras	Hasta	Procedimiento
Si	Hacer	Repetir	Incremento	Función
Entonces	Caso	Hasta que	Cada	Imprimir
Leer	Retornar			

Cuadro 1.3
Tipos de datos

Carácter	Cadena	Entero	Real	Booleano

1.4.2. Estructuras de control

Tal como se ha comentado en el apartado anterior, el pseudocódigo utiliza las estructuras de control propias de la programación estructurada. Por este motivo se emplearán secuencias que representen las estructuras de control *secuencial*, *alternativa* e *iterativa*.

A) Estructuras de control secuencial

Describen bloques de instrucciones que son ejecutadas en orden de aparición (secuencialmente). Los bloques pueden estar delimitados por las expresiones `Inicio-Fin` o bien estar contenidos en otras estructuras. A continuación, se muestra un ejemplo de estructura secuencial:

```
Inicio
 <instrucción1>
 …
 <instrucciónN>
Fin
```

B) Estructuras de control alternativa

La estructura de control *alternativa* o *selectiva* encauza el flujo de ejecución hacia un bloque de instrucciones u otro en función de la evaluación que se realiza sobre una condición determinada.

Hay diferentes subtipos de este tipo de estructura de control:

1. *Alternativa simple*: establece un conjunto de instrucciones que se ejecutarán si se cumple una condición que retornará un valor *booleano*. Ejemplo:

```
Si <condición> entonces
 <instrucción1>
     …
 <instrucciónX>
Fin Si
```


2. *Alternativa doble*: añade otro bloque de instrucciones que se ejecuta en caso de que no se cumpla la condición. Ejemplo:

```
Si <condición> entonces
 <instrucciones1>
Si no
 <instrucciones2>
Fin Si
```

3. *Alternativa múltiple*: permite ejecutar diferentes bloques de instrucciones según el valor que tome una expresión que es comparada con los valores de cada caso o bien mediante el anidamiento de diferentes estructuras de alternativa doble cuyas condiciones son excluyentes. Ejemplo de bloques equivalentes en resultado:

Equivalentes

```
Según <expresión> hacer
   Caso <valor1>
     <instrucciones1>
   Caso <valor2>
     <instrucciones2>
   Caso <valor3>
     <instrucciones3>
   Otro caso
     <instruccionesN>
Fin Según
```

```
Si <condición1> entonces
 <instrucciones1>
Si no
   Si <condición2> entonces
     <instrucciones2>
   Si no
       Si <condición3> entonces
        <instrucciones3>
       Si no
        <instruccionesN>
       Fin Si
   Fin Si
Fin Si
```

C) Estructuras de control iterativa

La estructura de control *iterativa* permite que un bloque de instrucciones sea ejecutado mientras se cumpla una condición.

Hay diferentes subtipos de este tipo de estructura de control:

1. *Iteración con salida al principio (while)*: primeramente, evalúa la condición y en caso de cumplirse ejecuta el bloque de instrucciones. La condición deberá cambiar de valor según las instrucciones contenidas para evitar bucles infinitos. Es posible que nunca llegue a ejecutarse el bloque de instrucciones. Ejemplo:

```
Mientras <condición> Hacer
 <instrucciones>
Fin Mientras
```

2. *Iteración con salida al final (repeat y do while):* primeramente, ejecuta el bloque de instrucciones y posteriormente evalúa la condición. La condición deberá cambiar de valor según las instrucciones contenidas para evitar bucles infinitos. El bloque de instrucciones se ejecutará, como mínimo, una vez. La versión que hace uso de *repeat* ejecutará las instrucciones hasta que se cumpla la condición mientras que la variante *do while* ejecutará las instrucciones mientras se cumpla la condición. A continuación, se muestran ejemplos de pseudocódigo para *repeat* y *do while*:

```
Repetir
 <instrucciones>
Hasta Que <condición>
```

```
Hacer
 <instrucciones>
Mientras <condición>
```

3. *Iteración con contador (for):* ejecutará el bloque de instrucciones un número determinado de iteraciones. Hace uso de una variable que irá incrementando o decrementando por cada iteración hasta que se cumpla la condición de salida. La condición de control, que se traducirá a `i==N` o `i==1` según sea el incremento, deberá ser falsa para que sigan ejecutándose las instrucciones. Ejemplos de *for incremental* y *decremental:*

```
entero i
Para i ← 1 Hasta N Incremento 1 Hacer
 <instrucciones>
```

Incremento positivo

```
entero i
Para i ← N Hasta 1 Incremento -1 Hacer
 <instrucciones>
```

Incremento negativo

4. *Iteración para cada (for each):* ejecutará el bloque de instrucciones para cada elemento contenido en un conjunto. Ejemplo:

```
entero i
Para Cada elemento En conjunto Hacer
 <instrucciones>
Fin Para Cada
```

D) Estructuras modulares

También es posible describir mediante pseudocódigo la descomposición modular de los programas representando los procedimientos y las funciones encargados de realizar las tareas.

1. Procedimientos

Cada bloque de código contenido en un procedimiento se ocupará de llevar a cabo un conjunto de instrucciones cuando este sea invocado. Podrá recibir *argumentos* con los que trabajar a la hora de ser llamado si se ha definido el *parámetro* correspondiente, pero en ningún caso retornará un valor de salida. A continuación, se muestra un ejemplo de pseudocódigo para un procedimiento:

```
Procedimiento nombre (tipo parámetro1, tipo parámetro2, …)
 <instrucciones>
Fin Procedimiento
```

A continuación, se muestran dos ejemplos típicos de programación correspondientes a *HolaMundo* y a un programa *Saludo*:

```
Procedimiento HolaMundo ()
   imprimir ("Hola Mundo")
   imprimir ("----------")
Fin Procedimiento

Inicio
   HolaMundo ()
Fin
```

```
Procedimiento Saludo (cadena nombre)
   imprimir ("Hola" + nombre)
Fin Procedimiento
Inicio
   cadena nombre
   imprimir ("Introduzca su nombre: ")
   leer (nombre)
   Saludo (nombre)
Fin
```

2. Funciones

Al igual que los procedimientos, una función podrán recibir *argumentos* con los que trabajar a la hora de ser llamada si se ha definido el *parámetro* correspondiente, pero, a diferencia de los procedimientos, retornará un valor de salida. Ejemplo de pseudocódigo para una función:

```
Función nombre (tipo parámetro1, tipo parámetro2, …) : tipoRetorno
 <instrucciones>
 …
   retornar X
Fin Función
```

Recuerda

✓ El pseudocódigo es un lenguaje que permite realizar una aproximación muy cercana a la implementación de un problema determinado en un lenguaje real de programación (C, Pascal, Algol…).

No se ajusta a ningún estándar y, por lo tanto, puede presentar diferencias dependiendo de qué programador lo escriba.

A continuación, se muestra un ejemplo típico de programación:

```
Función TextoParImpar (entero número) : cadena
   cadena resultado
   Si (número % 2 = 0) entonces
        resultado ← "par"
   Si no
        resultado ← "impar"
   Fin Si
   retornar resultado
Fin Función

Inicio
   entero número
   imprimir ("introduzca el número …")
   leer (número)
   imprimir ("el número introducido es " + TextoParImpar (número))
Fin
```

Declaración de la función

Llamada a la función

Actividad propuesta 1.4

Empleando pseudocódigo desarrolla un programa que solicite la introducción de dos números por teclado (*núm. 1* y *núm.* 2) y muestre por pantalla la suma, resta, multiplicación y división entera (siendo *núm.* 2 el divisor). Ten en consideración que si *núm.* 2 es 0, el resultado será infinito (∞).

1.5. Diagramas de flujo

Un *diagrama de flujo*, *ordinograma* o *flujograma* es una representación gráfica de un algoritmo o proceso. Son utilizados en informática, aunque también se emplean en otros ámbitos diferentes.

Facilita la comprensión del algoritmo gracias a la descripción visual que aporta sobre el flujo de ejecución de este.

Para saber más

Otra técnica empleada para la representación de procesos son los diagramas *Nassi-Shneiderman o Chapin*. Este tipo de diagramas permite representar, de forma análoga a los diagramas de flujo, el flujo de ejecución de un programa o algoritmo.

1.5.1. Simbología

Los diagramas de flujo se construyen utilizando un conjunto de símbolos que se van enlazando entre sí para dar significado al proceso. En el cuadro 1.4 se muestra cada uno de los símbolos empleados, así como una descripción de estos:

- ✓ Todo diagrama de flujo comienza y finaliza con un *terminal* representado mediante un óvalo o elipse.
- ✓ El trapecio rectángulo representa la *entrada de datos desde teclado*. Indica la detención del proceso a la espera de que el usuario teclee los datos.
- ✓ La comunicación con los periféricos para la *entrada* o *salida* de datos se representa mediante un paralelogramo.
- ✓ En todo algoritmo existe un orden en el que se realizan sus operaciones. En los diagramas de *flujo* ese orden se indica mediante una flecha.
- ✓ Las *decisiones* se representan mediante un rombo del que salen tantas líneas de flujo como alternativas sean posibles.
- ✓ Si el algoritmo a representar es muy grande, puede ser complejo diseñar su diagrama en un único bloque y lo más conveniente será dividir el diagrama en bloques más pequeños. Para ello se pueden emplear *conectores* que representarán el punto al cual saltar una vez se llegue a ese punto. Los saltos se representarán mediante una circunferencia si son en la misma página, o con un pentágono irregular si son en otra.
- ✓ Los *procesos* que llevan a cabo las operaciones internas de cálculo se representan mediante un rectángulo.
- ✓ Es posible que haya determinadas tareas que, por su complejidad, no puedan ser representadas como un proceso. En ese caso serán descompuestas en *subprocesos* que estarán definidos en otro lugar y que están representados por un rectángulo con doble línea en cada lado.
- ✓ Las *bases de datos* con las que se intercambia información tienen su propio símbolo (cilindro).
- ✓ Los *documentos impresos* también tienen sus propios símbolos de representación.

Cuadro 1.4
Simbología propuesta por la American National Standards Institute (ANSI)

Hasta este punto se han representado los símbolos más comunes. Se ha considerado innecesario describir la simbología al completo debido a que algunos de los símbolos existentes han quedado obsoletos al representar elementos que ya no se utilizan (cintas magnéticas, tarjetas perforadas…).

1.5.2. Estructuras de control

Mediante los diagramas de flujo se podrán representar las estructuras de control anteriormente descritas en pseudocódigo.

Figura 1.15
Secuencial.

Figura 1.16
Alternativa simple.

Figura 1.17
Alternativa doble.

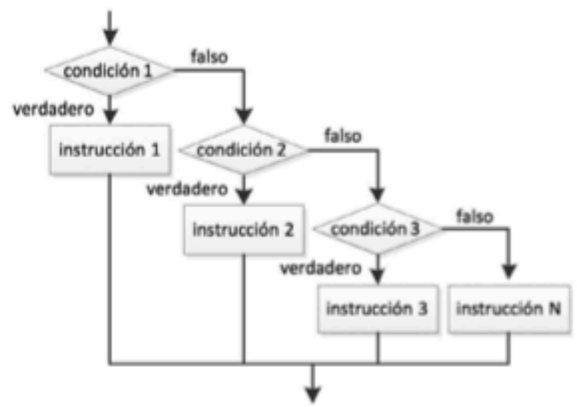


Figura 1.18
Alternativa múltiple con simples anidadas.

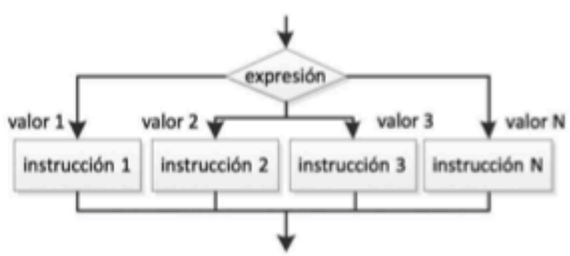


Figura 1.19
Alternativa múltiple II.

Figura 1.20
Iterativa *while*.

Figura 1.21
Iterativa *repeat*.

Figura 1.22
Iterativa *do while*.

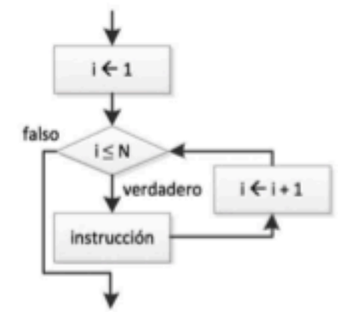


Figura 1.23
Iterativa *for incremental*.

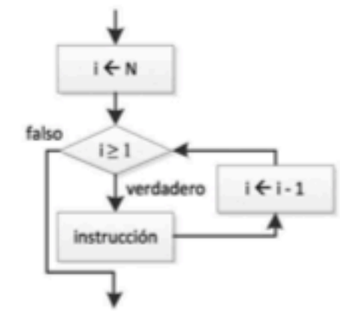


Figura 1.24
Iterativa *for decremental*.

Cada función o procedimiento (estructuras modulares) quedará descrito por un diagrama independiente y la invocación a dicho subprograma se realizará desde el diagrama principal.

- En el caso de invocar un procedimiento, lo representaremos mediante una llamada al subproceso que queda identificado por el rectángulo de lados dobles. A continuación, se muestran los diagramas correspondientes a los ejemplos realizados en pseudocódigo:

Figura 1.25
Ejemplo *HolaMundo*.

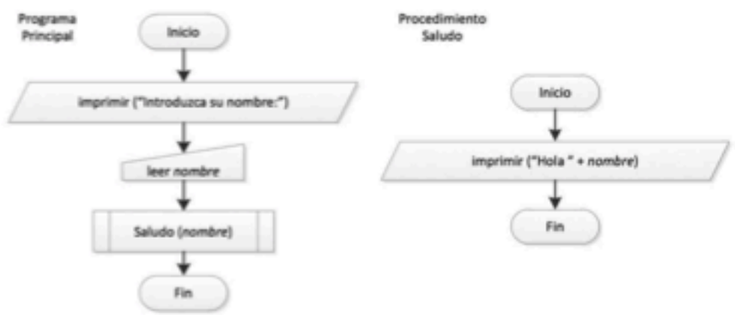


Figura 1.26
Ejemplo *Saludo*.

- Para invocar una función bastará con indicar la llamada en el proceso:

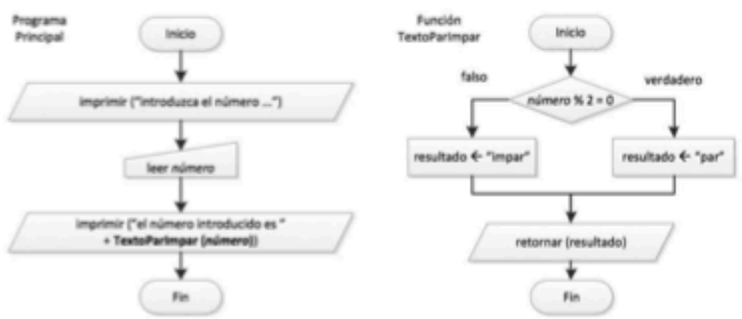


Figura 1.27
Ejemplo de función.

Recuerda

✓ Los diagramas de flujo son la representación gráfica de un algoritmo, proceso o programa.
La simbología empleada será la propuesta por ANSI.

Actividad propuesta 1.5

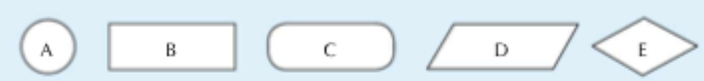
Realiza el diagrama de flujo para un programa similar al del enunciado 1.4, en el que se solicite la introducción de dos números por teclado (*num1* y *num2*) y se muestre por pantalla la suma, resta, multiplicación y división entera (siendo num2 el divisor). Ten en cuenta que si num2 es 0, el resultado será infinito (∞).

Resumen

- Existen varios paradigmas de programación que sirven de guía para el desarrollo de programas; la programación estructurada, la programación modular y la programación orientada a objetos son varios de ellos.
- El empleo de más de uno de esos paradigmas se conoce como programación multiparadigma y a menudo se presenta como la solución más adecuada en la mayor parte de los problemas.
- Para llevar a cabo la descripción de los algoritmos se dispone de varias técnicas. Entre ellas se encuentran el pseudocódigo, diagramas de flujo y diagramas Nassi-Shneiderman.
- El pseudocódigo representa un código universal para implementar un programa o algoritmo.
- Los diagramas de flujo y los diagramas Nassi-Shneiderman son técnicas gráficas de representación que permiten visualizar fácilmente el flujo de ejecución de un programa o algoritmo.

Ejercicios propuestos

1. Define el concepto de *paradigma de programación*.
2. Describe las diferencias existentes entre una *función* y un *procedimiento*.
3. ¿Qué diferencia hay entre *cohesión* y *acoplamiento*?
4. Enumera y describe los tipos de estructuras de control que son empleados en programación estructurada.
5. ¿En qué consiste el concepto *divide y vencerás* que permite poner en práctica un proceso de desarrollo *top-down*?
6. Identifica y describe el significado de los siguientes símbolos:



Supuestos prácticos

1. Implementa mediante pseudocódigo un algoritmo que determine si un año es bisiesto. NOTA: un año es bisiesto si es divisible entre 4, a excepción de aquellos años que son divisibles entre 100 pero no entre 400.

2. Realiza la instalación de una aplicación que te permita realizar diagramas de flujo. Una alternativa muy utilizada es *Microsoft Visio*, que es con la que se han realizado los diagramas del capítulo. También se podrá instalar alguna alternativa libre como *OpenOffice Draw* o *Dia Diagram Editor*, o alguna herramienta de diseño online como *Lucidchart*.
3. Empleando la herramienta del supuesto práctico anterior, realiza el diagrama de un algoritmo que determine si un año es bisiesto.
4. Completa los apartados 1 y 3 creando un programa principal que sea el encargado de solicitar al usuario el año del cual se pretende conocer si es o no bisiesto y que invoque el algoritmo representado en los apartados anteriormente mencionados para conocer el resultado.

Actividades de autoevaluación

1. ¿Quién es considerado el creador o creadora del primer algoritmo informático?
   - ☐ *a)* Jacques Vaucanson.
   - ☐ *b)* Ada Byron.
   - ☐ *c)* Charles Babbage.
   - ☐ *d)* Basile Bouchon.
2. Corrado Böhm y Giuseppe Jacopini…
   - ☐ *a)* … proponen la utilización de sentencias de salto incondicional.
   - ☐ *b)* … redactaron el teorema del programa estructurado.
   - ☐ *c)* Las opciones *a)* y *b)* son correctas.
   - ☐ *d)* Todas las opciones son incorrectas.
3. Las instrucciones de salto incondicional (*go to*):
   - ☐ *a)* Son implementadas en gran parte de los lenguajes de programación.
   - ☐ *b)* Están desaconsejadas porque suelen contribuir al código espagueti.
   - ☐ *c)* Las opciones *a)* y *b)* son correctas.
   - ☐ *d)* Todas las opciones son incorrectas.
4. Según la escala de cohesión de Stevens, Myers, Constantine y Yourdon, el orden deseable de aparición de cohesión es:
   - ☐ *a)* Funcional, secuencial, procedural, temporal, lógica, comunicacional, casual.
   - ☐ *b)* Funcional, secuencial, comunicacional, procedural, temporal, lógica, casual.
   - ☐ *c)* Funcional, comunicacional, secuencial, procedural, temporal, lógica, casual.
   - ☐ *d)* Funcional, lógica, secuencial, comunicacional, procedural, temporal, casual.

5. De las siguientes afirmaciones selecciona aquella que sea falsa:
   - ☐ *a)* El tamaño ideal de un módulo es aquel que permita que sea visualizado en una pantalla (idealmente 30-50 líneas de código).
   - ☐ *b)* Según la simbología ANSI, el rombo es empleado para representar procesos.
   - ☐ *c)* La sintaxis del pseudocódigo varía dependiendo del programador que lo escriba.
   - ☐ *d)* Los procedimientos son un tipo de módulos empleados en programación estructurada.
6. Las estructuras de control *repeat* y *do while:*
   - ☐ *a)* Son exactamente la misma estructura de control, pero con distinto nombre.
   - ☐ *b)* Ejecutarán el bloque de instrucciones contenido un número determinado de iteraciones.
   - ☐ *c)* Ejecutarán el bloque de instrucciones contenido para cada elemento de un conjunto.
   - ☐ *d)* Ejecutarán el bloque de instrucciones contenido mínimo una vez.
7. Según el teorema del programa estructurado, un programa puede escribirse empleando únicamente:
   - ☐ *a)* Las estructuras secuencial, condicional y alternativa.
   - ☐ *b)* Las estructuras iterativa, incondicional y condicional.
   - ☐ *c)* Las estructuras secuencial, alternativa e iterativa.
   - ☐ *d)* Las estructuras incondicional, alternativa e iterativa.
8. Selecciona cuál de las siguientes es una opción correcta para llevar a cabo la modularización de un programa:
   - ☐ *a)* En diseños estructurados y a un alto nivel, serán las unidades.
   - ☐ *b)* En diseños estructurados y a más bajo nivel, serán los procedimientos y las funciones.
   - ☐ *c)* En diseños orientados a objetos y a un alto nivel, serán los paquetes y librerías.
   - ☐ *d)* Todas las opciones son correctas.
9. ¿Cuál de las siguientes no es un tipo de cohesión deseable?
   - ☐ *a)* Cohesión funcional.
   - ☐ *b)* Cohesión lógica.
   - ☐ *c)* Cohesión secuencial.
   - ☐ *d)* Cohesión comunicacional.
10. Los tipos de acoplamiento ordenados de más débil a más fuerte son:
    - ☐ *a)* Patológico, externo, global, normal.
    - ☐ *b)* Patológico, global, externo, normal.
    - ☐ *c)* Normal, externo, global, patológico.
    - ☐ *d)* Externo, normal, global, patológico.

SOLUCIONES:

1.	a	**b**	c	d	5.	a	**b**	c	d						
2.	a	**b**	c	d	6.	a	b	c	**d**	9.	a	**b**	c	d	
3.	a	b	**c**	d	7.	a	b	**c**	d	10.	a	b	**c**	d	
4.	a	b	**c**	d	8.	a	b	c	**d**						

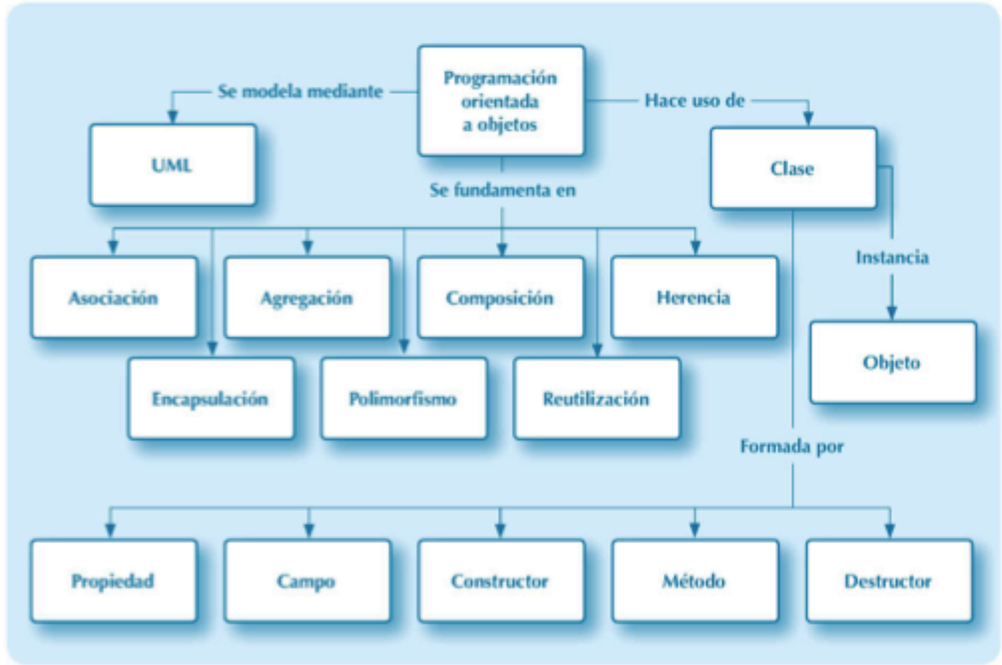
2

Programación orientada a objetos

Objetivos

- ✓ Definir el paradigma de programación orientada a objetos.
- ✓ Conocer los elementos que se emplean en programación orientada a objetos.
- ✓ Comprender la simbología básica de UML para representar objetos y sus relaciones.
- ✓ Familiarizarse con los fundamentos de la programación orientada a objetos.

Mapa conceptual



Glosario

Abstracción. Mecanismo de reducción de la complejidad que minimiza la información de los objetos a representar únicamente a los aspectos importantes.

Algoritmo. Conjunto de instrucciones que, expresadas según determinada lógica y orden, permiten obtener la solución a un problema determinado.

Clase abstracta. Clase que obvia determinados detalles y que por tanto no es completamente funcional. Se emplea con la finalidad de implementar relaciones de herencia y polimorfismo.

Clase ancestra. También conocida como clase base, clase padre o superclase, es la clase que define los campos, métodos y propiedades que otras clases pueden heredar.

Clase derivada. También conocida como clase descendiente, clase hija o clase extendida, es la clase que hereda los campos, métodos y propiedades de otras clases.

Clase. Elemento básico de programación en un paradigma orientado a objetos, que describe un posible conjunto de datos y un conjunto de operaciones a realizar sobre estos.

Encapsulación. Agrupamiento de los datos y las operaciones relacionadas con ellos en un mismo elemento.

Implementar. Acción de materializar y poner en funcionamiento una idea o proyecto concebido para dar solución a un determinado problema o necesidad.

Instanciar. Es la acción de crear instancias de una clase contenedora de información y de los métodos necesarios para manejarla.

Miembro. Conjunto de elementos pertenecientes a una clase que permiten establecer su información y su comportamiento.

Polimorfismo. Propiedad de los objetos que les permite adoptar diferentes formas en tiempo de ejecución.

POO. Paradigma de programación orientada a objetos.

UML. Abreviatura de *Unified Modeling Language,* es un lenguaje de modelado estandarizado consistente en un conjunto integrado de diagramas que sirven de ayuda al desarrollo de software.

2.1. Introducción

Al igual que la programación estructurada, la *programación orientada a objetos* (*POO*) es un tipo de programación que desciende del paradigma de programación imperativa. Tiene sus orígenes en el lenguaje de programación Simula 67, que fue el primero en introducir los conceptos *objeto* y *clase* en el año 1962.

En este capítulo se introducirá el paradigma de programación orientada a objetos, describiendo los diferentes elementos que lo conforman e introduciendo pequeños conceptos de diseño mediante el lenguaje unificado del modelado (UML de ahora en adelante) que facilitan la comprensión a la vez que permiten ilustrar los ejemplos.

Sabías que...

UML define muchos tipos de diagramas que facilitan el diseño de aplicaciones software basándose en la orientación a objetos.

Entre esos diagramas se encuentran aquellos que permiten representar la estructura estática del sistema, la interacción entre los diferentes elementos de este y los diferentes estados por los que puede atravesar cada uno de esos elementos.

Actualmente se utiliza la versión 2.5 de UML.

2.2. Auge de la programación orientada a objetos

La POO se ha convertido en el paradigma de programación más empleado desde la década de los ochenta y eso se debe a las muchas e importantes ventajas que aporta con respecto a otros paradigmas para el desarrollo.

- Por un lado la POO permite enfocar los problemas a resolver de una forma mucho más intuitiva de lo que lo hacen otros paradigmas. Mientras que en la programación estructurada datos y código están totalmente separados, en la POO se encuentran mezclados. En este paradigma se persigue trabajar con un modelo más cercano al mundo real en el que los diferentes objetos interaccionan entre sí. Implica un cambio en la manera de enfocar el desarrollo y de diseñar el código, más aún si se viene de un modelo puro de programación estructurada.
- Este tipo de programación viene a solventar las carencias que la programación estructurada presenta a la hora de afrontar el desarrollo de programas de gran envergadura. La programación orientada a objetos permite generar programas modulares y muy escalables.
- Con la creciente necesidad de desarrollar programas con interfaces gráficas muy depuradas, la POO consiguió convertirse en la mejor opción para conseguir una buena experiencia de usuario. Su capacidad para permitir reutilizar código sin necesidad de reescribirlo nuevamente facilitó mucho la tarea de los programadores, que vieron en esta tendencia una gran oportunidad de rentabilizar el esfuerzo realizado.

2.3. Características de la programación orientada a objetos

El paradigma de POO supuso una revolución en el mundo de la programación al incluir novedosos conceptos que permitían generar código de más calidad, facilitar el desarrollo y reutilizar de una forma más práctica el código desarrollado.

De la misma forma que la POO partió de paradigmas de programación existentes, otros paradigmas han tomado la POO como punto de partida para ser desarrollados. Con la creciente demanda de aplicaciones que ofreciesen una interfaz gráfica cada vez más sofisticada, surgió la necesidad de adaptar la POO a un tipo de programación que permitiese controlar los eventos que se producían sobre las interfaces gráficas con las que el usuario interactuaba.

A continuación, serán descritos cada uno de los elementos y características de este extendido paradigma.

2.3.1. Clase

La *clase* es el elemento básico de programación en un paradigma orientado a objetos. En ella se describe un posible conjunto de datos (*campos*, *propiedades* o ambos, según el lenguaje de programación empleado), un conjunto de operaciones a realizar sobre ellos (*métodos*), *constructores* y, posiblemente, *destructores*. Todos estos elementos son conocidos como *miembros*.

A partir de una clase se podrán instanciar tantos objetos de esta como sean necesarios; se entiende pues que una clase es una plantilla a partir de la cual definir y crear objetos.

Las clases se representan en UML en diagramas de clase mediante una caja dividida en tres áreas en las que se especifican el *nombre* de la clase y sus *miembros*:

- ✓ *Área superior:* muestra el *nombre* de la clase representada (*Persona*).
- ✓ *Área intermedia:* contiene los *campos* de la clase (*DNI, nombre, apellido1, apellido2, fechaNacimiento*).
- ✓ *Área inferior:* recoge los diferentes *métodos* de la clase (en este ejemplo únicamente el método constructor *Persona*).

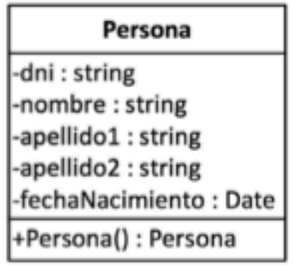


Figura 2.1
Representación de una clase.

Además de la información anteriormente descrita es muy común encontrar información relativa a la visibilidad de cada uno de los miembros de la clase.

Investiga

En UML existen varios tipos de diagramas. Investiga cómo construir un diagrama de clases representando estos elementos y las relaciones existentes entre ellos.

A) Campo

Se denomina *campo* a un elemento que se comporta como una variable propia de cada instancia de una clase y que define un atributo de un objeto.

En él se pueden almacenar datos, y sus valores pueden ser inicializados, modificados y leídos directamente sin ningún tipo de acción adicional; esto último dependiendo siempre en última instancia de la visibilidad establecida para el campo.

Un campo puede ser declarado de cualquier tipo incluyendo otras clases y su inicialización puede realizarse a la vez que son declarados o en su constructor.

B) Propiedad

Una *propiedad*, al igual que un campo, define un atributo de un objeto. Pero mientras que un campo dispone de un área de almacenamiento directamente accesible, la propiedad no tiene asociada una reserva de espacio de memoria para su almacenamiento. En realidad, permite acceder a un área de almacenamiento a través del campo definido previamente o bien a través de un método creado para tal efecto. Representa, por tanto, un mecanismo indirecto y flexible de acceso que permite, si así se desea, establecer una serie de acciones que serán llevadas a cabo de forma subyacente cada vez que se haga uso de esta para acceder a los datos, proporcionando un mayor control en el acceso.

Generalmente las propiedades son declaradas con un nombre representativo del dato que van a almacenar, y los métodos que se emplean para la lectura o modificación de los datos a través de propiedades son declarados con los prefijos *get* o *set* para la lectura o modificación de los datos, respectivamente.

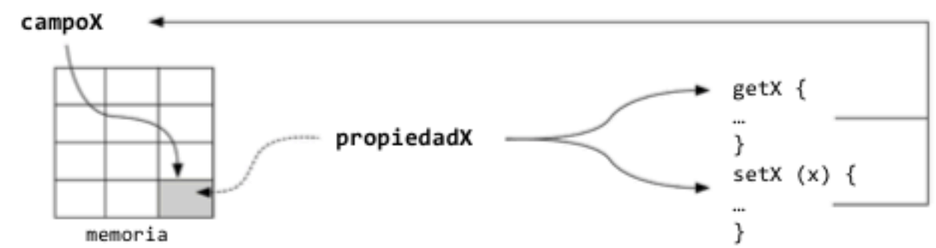


Figura 2.2
Acceso mediante campos versus propiedades.

La declaración de propiedades puede ser muy diferente dependiendo del lenguaje de programación que se esté utilizando (Delphi, C#…). Algunos lenguajes, como Java, pueden incluso no ofrecer mecanismos específicos para implementarlas.

C) Método

El *método* es la implementación de un algoritmo aplicable a todas las instancias de la clase y que puede, o no, utilizar los datos contenidos en ella.

Un método tomará forma de:

- *Función*, si al ser invocado el algoritmo contenido en ella ejecuta una serie de instrucciones y finalmente retorna un valor.
- *Procedimiento*, si al ser invocado el algoritmo contenido en él realiza una serie de acciones basadas generalmente en un conjunto de parámetros o argumentos. A diferencia de las funciones, un procedimiento no retorna ningún valor.

Que un método se implemente como función o procedimiento depende de varios factores. Primeramente, del lenguaje con el que se esté trabajando; mientras que algunos lenguajes como Java únicamente utilizan funciones, otros como Delphi hacen distinción entre procedimientos

y funciones. En segundo lugar, dependerá de cuál sea el comportamiento que se desee implementar; si únicamente se persigue realizar una serie de acciones sin retornar ningún valor, es preferible implementarlo como procedimiento.

D) Constructor

Un *método constructor* es un tipo especial de método perteneciente a una clase concreta y que se utiliza para instanciar objetos de esta.

Para cada clase pueden declararse varios métodos constructores, aunque también puede darse el caso de que no se declare ninguno. El hecho de que no se declare ningún método constructor no implica que no exista; por defecto, todas las clases disponen de un método constructor que, en caso de invocarse, inicializará los campos de dicha clase a los valores por defecto.

Actividad propuesta 2.1

¿Qué es un constructor parametrizado? Desarrolla tu respuesta.

E) Destructor

Un *método destructor* se define como aquel método encargado de destruir un objeto y liberar los recursos que tuviese asignados.

Una clase podrá tener un número indeterminado de destructores, aunque al igual que sucede con los constructores puede que no se declare ninguno. Que esto ocurra no implica que no exista; las clases disponen de un método destructor por defecto.

Será responsabilidad del programador hacer uso de los destructores en aquellas tecnologías en las que no existan mecanismos de destrucción automática de objetos.

Sabías que...

En algunos lenguajes, como Java, no existen los métodos destructores.

2.3.2. Objeto

Un *objeto* es una instancia de una clase determinada. Dicho de otra manera, se puede decir que un objeto es el resultado de crear en tiempo de ejecución un conjunto de datos que se almacena en memoria y que tiene una estructura como la que se describe en su clase.

Los campos, propiedades y métodos pertenecientes a un objeto pueden estar declarados en una única clase o bien en varias, en cuyo caso se tratará de un objeto que hereda los campos de otra clase ancestra. El concepto de *herencia* será descrito más adelante con detalle.

Los objetos pueden ser *simples* o *compuestos*. Un objeto simple estará formado por datos primitivos (un carácter, un dígito, un booleano…), mientras que un objeto compuesto estará formado por datos primitivos y por otros objetos pertenecientes a otras clases. El concepto de *composición de clases* será descrito más adelante con mayor detalle.

El hecho de que dos objetos tengan los mismos datos no significa que sean el mismo objeto. Cada uno tiene reservado un espacio único en memoria y, por tanto, es único e independiente.

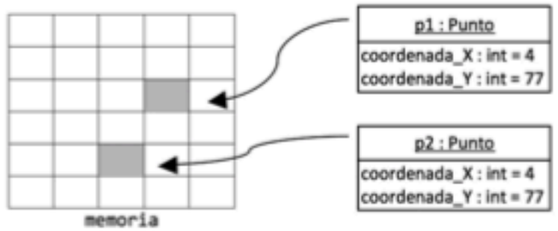


Figura 2.3
Diferentes objetos con mismos valores.

Investiga

Busca información sobre los diagramas de objetos y sobre cómo estos permiten representar los objetos de una aplicación.

Actividad propuesta 2.2

¿Es posible que un objeto contenga objetos de su misma clase? Justifica tu respuesta.

2.3.3. Encapsulación

Se define la *encapsulación* como el agrupamiento de los datos y las operaciones relacionadas con ellos en un mismo elemento (la clase ofrece este nivel de encapsulación). Los datos encapsulados deben ser convenientemente ocultados para que únicamente se pueda acceder a su información a través de interfaces bien definidas.

2.3.4. Abstracción

La *abstracción* es un mecanismo de reducción de la complejidad que minimiza la información de los objetos a representar únicamente a los aspectos más relevantes. Es importante

conseguir el nivel de abstracción adecuado que permita obviar aspectos insustanciales para el resultado que, de ser tenidos en cuenta, únicamente dificultarían el desarrollo y penalizarían el rendimiento.

2.3.5. Reutilización o reusabilidad

El modelo orientado a objetos permite reutilizar de forma fácil el código que ya se ha escrito anteriormente sin la necesidad de duplicarlo. La reutilización no solo es una posibilidad, sino que debería ser una máxima. A mayor reutilización, menor cantidad de líneas de código, mayor facilidad de generación de programas, menor posibilidad de cometer errores, menor dificultad de mantenimiento… Todo esto finalmente se consigue mediante el empleo de técnicas de herencia y composición de objetos.

2.3.6. Visibilidad

Una vez se han abordado los conceptos *encapsulación* y *abstracción*, se hace necesario describir también el concepto de *visibilidad*. Este nuevo concepto establece la forma en la que los miembros de una clase serán accedidos desde otros puntos del programa. Toma vital importancia no solo a la hora de implementar los algoritmos, sino también a la hora de crear relaciones de herencia entre diferentes clases.

Distinguiremos los siguientes niveles de visibilidad:

- *Privada*: se representa mediante el signo menos (−). Los miembros declarados con visibilidad privada únicamente serán visibles desde la misma clase en la cual se han declarado.
- *Pública*: se representa mediante el signo (+). Los miembros declarados con visibilidad pública serán visibles desde cualquier lugar del programa.
- *Protegida*: se representa mediante el signo (#). Los miembros declarados con visibilidad protegida únicamente serán visibles desde la misma clase en la cual se han declarado y desde las clases que hereden de ella.

Dependiendo del lenguaje de programación que se utilice para llevar a cabo la implementación del código, es posible encontrar algún nivel más.

Para saber más

Algunas tecnologías de programación, como .NET o Java, disponen de más niveles de visibilidad.

2.3.7. Relaciones entre clases

A) Herencia

La *herencia* es una propiedad exclusiva de las clases que permite la declaración de nuevas clases a partir de otras declaradas previamente. La clase origen, desde la que se hereda, se denomina *clase ancestra*, *clase base*, *clase padre* o *superclase*; la clase que hereda se denomina *clase derivada*, *clase extendida*, *clase hija* o *subclase*.

Cuando la clase B "hereda" de la clase A, todos los miembros de la clase A son transferidos a la clase B, a excepción de constructores y destructores. La clase B podrá añadir miembros propios de la nueva clase (campos, métodos y propiedades) o sobrescribir los miembros heredados.

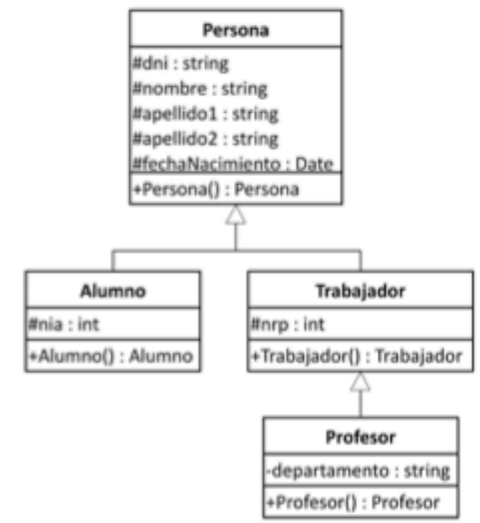


Figura 2.4
Ejemplo de representación de la herencia.

Aunque según el paradigma de la POO la herencia puede ser múltiple, lo cual implica que para una misma clase pueda haber más de una clase padre desde la cual derivar al mismo tiempo, en realidad son pocos los lenguajes de programación que implementan la herencia múltiple (C++ o Eiffel son algunos ejemplos de lenguajes que sí implementan la herencia múltiple).

Ligado al concepto de herencia está también el de *visibilidad*. Aunque todos los miembros son heredados, excepto constructores y destructores, únicamente serán accesibles desde la clase hija aquellos que hayan sido declarados como miembros *públicos (+)* o *protegidos (#)*. Este concepto será visto con más detalle en apartados posteriores.

B) Asociación, composición y agregación

Además de las relaciones de generalización y herencia explicadas anteriormente, existen otros tipos de relaciones y todas ellas representan una conexión semántica entre los elementos de un modelo.

La *asociación* es un tipo de relación entre los objetos de una o varias clases que interactúan entre sí. Cada uno de los objetos puede existir por sí mismo y sin depender del otro. Se representa mediante una línea continua.

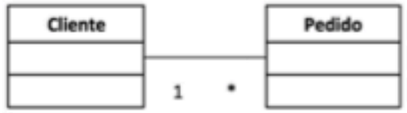


Figura 2.5
Asociación.

Otros tipos de asociaciones dedicadas a representar las relaciones existentes entre un objeto "compuesto" y objetos "componentes" son:

- *Agregación:* representa una asociación opcional, de manera que los componentes pueden existir por sí mismos e incluso formar parte de otras relaciones de composición con otros objetos. Las relaciones de agregación también son conocidas como composición débil. Se representa mediante un rombo sin fondo.

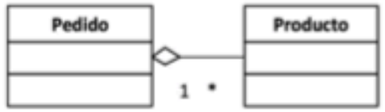


Figura 2.6
Agregación.

- *Composición:* representa una asociación obligatoria. Es decir, los componentes son totalmente dependientes del compuesto y no tiene sentido su existencia en ausencia de este. Por ello el objeto compuesto se debe hacer cargo de la destrucción de los componentes cuando es destruido. Se representa mediante un rombo con fondo negro.

Figura 2.7
Composición.

2.3.8. Polimorfismo

El *polimorfismo* es la propiedad de los objetos que les permite adoptar diferentes formas en tiempo de ejecución, así como que un objeto de una clase derivada sea utilizado como un objeto de la clase base.

Junto con la herencia, con la que está estrechamente relacionado, establece uno de los pilares de la programación orientada a objetos.

Puede encontrarse polimorfismo en diferentes formas:

- *Asignación polimorfa:* se presenta cuando se almacena en un elemento declarado con un tipo A (clase base) un objeto de un tipo B (clase derivada). Para ello el lenguaje debe permitir relajar el sistema de tipos.
- *Ejecución polimorfa:* se presenta cuando se invocan métodos del mismo nombre sobre instancias de diferente clase sin reparar en ello.

Recuerda

✓ En la programación orientada a objetos, los datos y los métodos que hacen uso de estos son declarados dentro de la misma clase.

La programación orientada a objetos es más adecuada para abordar proyectos de gran tamaño que el paradigma de programación estructurada.

Una *clase* es un molde o plantilla que define, en la fase de desarrollo, el aspecto (visibilidad), la composición (campos) y el comportamiento (propiedades y métodos) de los elementos que serán creados a partir de ella.

Los elementos que serán creados en tiempo de ejecución a partir de la definición de la clase son los *objetos*.

Cada clase puede definir varios *métodos constructores*, que serán empleados para llevar a cabo la instanciación de objetos (creación) a excepción de las *clases abstractas*, que son empleados con otros fines (*polimorfismo*, declaración de clases base a partir de las cuales realizar *herencia*...).

Actividad propuesta 2.3

Elabora un cuadro resumen con las características principales de la programación orientada a objetos.

Resumen

- Para llevar a cabo la descripción y el diseño de programas orientados a objetos se utiliza UML.
- Al igual que la programación estructurada, la programación orientada a objetos (POO) es un tipo de programación que desciende del paradigma de programación imperativa.

- La POO permite enfocar los problemas a resolver de una forma mucho más intuitiva de lo que lo hacen otros paradigmas.
- En la POO los datos (campos, propiedades o ambos) y el código que los utiliza (métodos, constructores y destructores) están mezclados en un elemento denominado *clase*.
- Las clases determinan la estructura que tendrán los objetos instanciados a partir de las mismas.
- La POO se basa en: encapsulación, abstracción, reutilización, visibilidad, herencia y polimorfismo.

Ejercicios propuestos

1. Define el concepto *UML*.
2. Describe las diferencias existentes entre el acceso a información a través de un campo respecto a una propiedad.
3. ¿Qué diferencia hay entre *agregación* y *composición*? Propón un ejemplo.
4. Enumera y describe los elementos de los que está formada una clase. ¿Cuál de ellos no está disponible en algunas tecnologías?
5. ¿En qué se diferencia una *clase* de un *objeto*? Razona tu respuesta.

Supuestos prácticos

1. Supón que debes modelar una aplicación para la Liga de Fútbol Profesional en la que se manejará información relativa a equipos de fútbol y jugadores. De cada equipo es necesario conocer su nombre y presupuesto. De cada jugador es necesario conocer el equipo al que pertenece, sus datos personales (DNI o NIE, nombre, apellidos, año de nacimiento…). Elabora en papel el diagrama de clases, prestando atención a los miembros que contendrá cada una de las clases, las relaciones existentes entre ellas y a los niveles de visibilidad de cada miembro.
2. Supón que debes modelar una aplicación para una compañía aérea en la que se manejará información relativa a aeronaves, aeropuertos, vuelos y localizaciones (ciudades de origen-destino). De cada aeronave es necesario almacenar el identificador del avión, número de pasajeros, autonomía máxima y modelo. Además, la compañía quiere distinguir también entre los aviones de hélice y los de turbinas, almacenando además en estos últimos el número de turbinas y modelo de los motores. De cada aeropuerto es necesario conocer el identificador, el nombre y la localización en la que se encuentra. De cada vuelo es necesario conocer los aeropuertos que conectará (suponiendo que no hace escala) así como la aeronave que lo realiza. De cada localización es necesario conocer la ciudad en la que se encuentra, dirección y coordenadas geográficas (latitud y longitud). Elabora en papel el diagrama de clases del supuesto anterior, prestando atención a los miembros que contendrá cada una de las clases, las relaciones existentes entre ellas y a los niveles de visibilidad de cada miembro.
3. Realiza la instalación de una aplicación que permita realizar diagramas de clases UML. Una alternativa muy utilizada es *Microsoft Visio*, que es con la que se han realizado los diagramas del capítulo. También se puede instalar alguna alternativa gratuita como *ArgoUML* o alguna herramienta de diseño online como *Lucidchart* o *draw.io*.
4. Realiza el diagrama del supuesto práctico 2 con la herramienta elegida en el supuesto práctico 3, teniendo en cuenta que algunos vuelos sí podrán hacer escala en un aeropuerto antes de llegar al destino.

Actividades de autoevaluación

1. Selecciona la afirmación correcta:
   - ☐ *a)* C++ y Java soportan herencia múltiple pero Eiffel no.
   - ☐ *b)* Java y Eiffel soportan herencia múltiple pero C++ no.
   - ☐ *c)* C++ y Eiffel soportan herencia múltiple pero Java no.
   - ☐ *d)* C++, Eiffel y Java soportan herencia múltiple.
2. La relación en la que un componente no tiene razón de ser en ausencia del compuesto es:
   - ☐ *a)* Asociación.
   - ☐ *b)* Composición.
   - ☐ *c)* Agregación.
   - ☐ *d)* Las opciones *b)* y *c)* son correctas.
3. ¿Cuál de las siguientes relaciones se denomina *composición débil*?
   - ☐ *a)* Asociación.
   - ☐ *b)* Herencia.
   - ☐ *c)* Agregación.
   - ☐ *d)* Todas las opciones anteriores son incorrectas.

4. Indica cuál de las siguientes afirmaciones sobre las propiedades es falsa:
   - ☐ *a)* Representa un atributo de un objeto.
   - ☐ *b)* Permite acceder al espacio de almacenamiento de forma flexible y personalizada.
   - ☐ *c)* Suele hacer uso de métodos *get* y *set* para la lectura y escritura de la información.
   - ☐ *d)* A diferencia de los campos y métodos, no se puede configurar su visibilidad.

5. Un método que tome forma de función:
   - ☐ *a)* Podrá definir varios parámetros de entrada y podrá devolver un valor.
   - ☐ *b)* Podrá definir varios parámetros de entrada y devolverá un valor.
   - ☐ *c)* Definirá varios parámetros de entrada y podrá devolver un valor.
   - ☐ *d)* Definirá varios parámetros de entrada y devolverá un valor.

6. Señala la afirmación correcta:
   - ☐ *a)* La abstracción es un mecanismo de reducción de la complejidad que reduce la información de los objetos a representar únicamente a los aspectos importantes.
   - ☐ *b)* La encapsulación es el agrupamiento de los datos y las operaciones relacionadas con ellos en un mismo elemento.
   - ☐ *c)* Los campos, propiedades y métodos pertenecientes a un objeto pueden estar declarados en una única clase o bien en varias.
   - ☐ *d)* Todas las opciones anteriores son correctas.

7. Si se desea que un miembro de una clase sea visible únicamente desde la misma clase y las clases que hereden de ella, ¿qué configuración se debe aplicar?
   - ☐ *a)* Pública (+).
   - ☐ *b)* Protegida (#).
   - ☐ *c)* Privada (–).
   - ☐ *d)* Todas las opciones anteriores son correctas.

8. Si la clase A hereda de B, y B hereda de C:
   - ☐ *a)* A también hereda de C.
   - ☐ *b)* Elementos declarados de tipo A podrán almacenar elementos declarados de tipo B.
   - ☐ *c)* Elementos declarados de tipo A podrán almacenar elementos declarados de tipo C.
   - ☐ *d)* Todas las opciones anteriores son incorrectas.

9. ¿Cuál de los siguientes no es un miembro de una clase?
   - ☐ *a)* Los campos.
   - ☐ *b)* El identificador.
   - ☐ *c)* Los métodos.
   - ☐ *d)* Los constructores y destructores.

10. Indica la afirmación falsa:
   - ☐ *a)* Un objeto es una instancia de una clase determinada.
   - ☐ *b)* Un objeto hereda los campos de otra clase ancestra.
   - ☐ *c)* Las opciones *b)* y *c)* son correctas.
   - ☐ *d)* Todas las opciones anteriores son incorrectas.

SOLUCIONES:

1.	a b **c** d	5. a **b** c d		
2.	a **b** c d	6. a b c **d**	9. a **b** c d	
3.	a b **c** d	7. a **b** c d	10. a b c **d**	
4.	a b c **d**	8. **a** b c d		

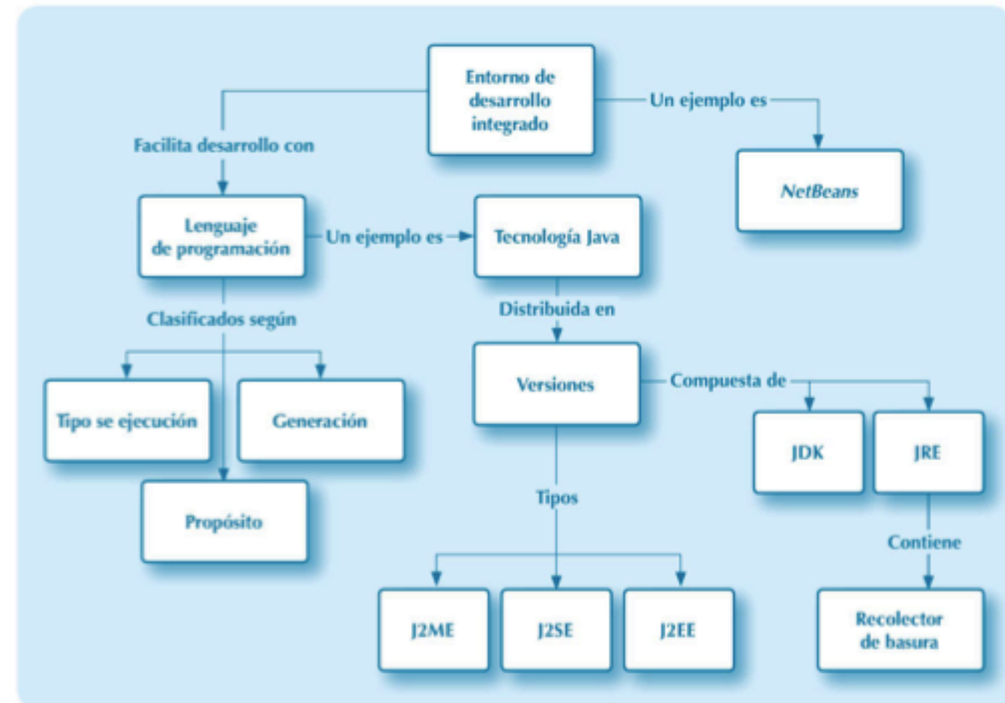
3

Lenguajes de programación y entornos de desarrollo integrado

Objetivos

- ✓ Conocer y clasificar los diferentes lenguajes de programación.
- ✓ Describir las características de la tecnología Java.
- ✓ Manejar las herramientas de la tecnología Java desde línea de comandos.
- ✓ Utilizar los entornos integrados de desarrollo.

Mapa conceptual



Glosario

Algoritmo. Conjunto de instrucciones que, expresadas según determinada lógica y orden, permiten obtener la solución a un problema determinado.

Código fuente. Programa escrito en un lenguaje de alto nivel, más cercano al lenguaje que entiende un humano que al que entiende un microprocesador, que finalmente es compilado para generar un código máquina que pueda ser ejecutado o interpretado.

Código máquina o código objeto. Conjunto de códigos que son interpretados directamente por un microprocesador para que sea ejecutado y que son dependientes de este último.

Compilador. Complejo programa encargado de la traducción de código fuente a código objeto o código máquina. El proceso de compilación se divide en varias fases: análisis léxico, análisis sintáctico, análisis semántico, generación del código intermedio, optimización del código, detección de errores, generación de código… Finalmente también hace uso de otros programas que son el enlazador y el ensamblador.

CORBA. Siglas de *Common Object Request Broker Architecture*. Estándar que permite a programas que se ejecutan en distintas ubicaciones físicas interactuar entre sí incluso cuando están desarrollados con diferentes lenguajes de programación.

Depurador. También conocido como *debugger*, es un módulo empleado para la detección de errores y la optimización del código fuente.

Enlazador. También conocido como *linker*, se ocupa de juntar el código objeto generado por el compilador, los recursos referenciados desde el código fuente y las bibliotecas utilizadas, eliminando los recursos no utilizados.

Ensamblador. Programa empleado para traducir código ensamblador en código máquina.

Heap. Zona de memoria dinámica en la que se almacenan los objetos que se van creando cuyas referencias son almacenadas en el STACK.

IDE. Entorno de desarrollo integrado que facilita la creación, mantenimiento y depuración de programas.

Índice TIOBE. Informe mensual que elabora y publica la empresa TIOBE Software BV sobre la cuota de mercado de los diferentes lenguajes de programación.

JSP. Acrónimo de *JavaServer Pages*. Tecnología Java empleada para el desarrollo de páginas web dinámicas cuyo contenido se genera mediante la inclusión de etiquetas con código en el código HTML.

JVM. Acrónimo de Java Virtual Machine (máquina virtual de Java).

Lenguaje de programación. Conjunto de signos que, ordenados conforme a determinadas reglas, permiten la escritura de programas. Aunque existen lenguajes de programación que, por sus características, pueden ser empleados para la creación de programas de muy diverso ámbito, otros están claramente dirigidos a la ejecución de tareas muy concretas.

Open source. Término traducido como *código abierto* o *fuente abierta*, que se refiere a aquello que puede ser modificado y compartido porque su contenido y diseño es de acceso público.

RMI. Acrónimo de *Java Remote Method Invocation*. Mecanismo que permite invocar a través de la red métodos pertenecientes a objetos que están ejecutándose en otra ubicación.

Servlet. Pequeño programa Java que se ejecuta en un servidor con la finalidad de atender peticiones y devolver respuestas de contenido dinámico.

Shell. Intérprete de comandos que sirve de interfaz entre el sistema operativo y el usuario.

Shortcut. Combinación de teclas que representa un atajo de teclado.

Stack. Zona de memoria estática en la que se almacenan las referencias a los objetos que se van creando y que se almacenan en el *Heap*.

Webservice. Conjunto de protocolos y tecnologías diseñadas para el intercambio de información entre diferentes máquinas a través de una red de telecomunicaciones.

Zip. Formato y algoritmo de compresión sin pérdida de información.

3.1. Introducción

Hoy en día los sistemas informáticos y los programas que sobre estos se ejecutan están presentes en todo lo que nos rodea. En muy poco tiempo se ha pasado de realizar una gran cantidad de tareas de forma manual a realizarse sobre programas informáticos. Pero no solo ha supuesto un cambio en los procesos ya existentes, sino que también han encontrado su sitio en nuevos escenarios. Todo ello promovido por las mejoras en las infraestructuras de telecomunicaciones, las innovaciones y el abaratamiento en los componentes de equipos informáticos, así como la mejora de las prestaciones de estos últimos.

Antes de llegar a este punto, ha sido necesario recorrer un largo camino en el que muchas tecnologías han sido desarrolladas y han evolucionado conforme a las necesidades. Como consecuencia de esta evolución se ha cambiado el enfoque con el que se analizan los problemas, se han mejorado las técnicas con las que se proponen sus soluciones y, por consiguiente, se han actualizado los lenguajes de programación a utilizar.

Lo que comenzó siendo una secuencia de ceros y unos, interpretados por un sistema para generar un resultado y que fue denominado *código máquina*, fue evolucionando hacia secuencias de palabras siguiendo cierto orden lógico que generaban el mismo resultado y que facilitaban el desarrollo de dichos programas. Podríamos hablar por tanto de distintas generaciones de lenguajes de programación.

3.2. Lenguajes de programación

Como ya se ha mencionado en el apartado introductorio, existe multitud de lenguajes de programación y la gran mayoría ha sufrido su propia evolución a lo largo del tiempo. No todos los lenguajes tienen las mismas características ni tampoco han sido creados para el mismo propósito. Es por ello por lo que se les puede clasificar en función de varios aspectos en distintas categorías:

- *Según generación:* distinguiéndose entre lenguajes de primera generación, segunda generación, tercera generación, cuarta generación y quinta generación.
- *Según su tipo de ejecución:* se pueden encontrar lenguajes interpretados y lenguajes compilados.
- *Según su propósito:* diferenciándose entre lenguajes de propósito general y de propósito específico.

Recurso digital

En el anexo web 3.1, disponible en www.sintesis.com y accesible con el código indicado en la primera página del libro, encontrarás información sobre la clasificación de los lenguajes de programación.

3.3. Java

Aunque no es la primera vez que se habla de Java en este libro, sí será en este apartado en el que se realizará una descripción detallada de qué es, en qué consiste y de por qué se ha elegido como tecnología vehicular sobre la cual aprender programación.

3.3.1. Descripción de la tecnología

Java es uno de los lenguajes de programación más populares, que se puede utilizar libremente y que cuenta con la mayor demanda en el mercado según el Índice TIOBE. Este potente lenguaje permite programar casi cualquier tipo de aplicación (juegos, aplicaciones web, aplicaciones de escritorio, servicios…) y está compuesto por varias tecnologías.

Sabías que…

Java fue diseñado inicialmente para utilizarse en electrodomésticos y, aunque no tuvo el éxito esperado, debido a su buen diseño y características, se decidió utilizarlo en el desarrollo de aplicaciones para ordenadores.

Tal fue el impacto de la tecnología Java en el mundo de la programación, que rápidamente desencadenó el nacimiento de nuevas tecnologías basadas en ella. *Microsoft Visual J++*, *J#* o *Processing* son algunas de las tecnologías que han surgido a partir de Java.

No debe confundirse Java con *JavaScript*, puesto que la única relación que guardan es la de su sintaxis.

Java fue anunciado públicamente el 23 de mayo de 1995, pero no fue hasta el 23 de enero del 1996 cuando se publicó la primera versión del JDK *(JDK 1.0)* de la mano de *Sun Microsystems*, adquirida hace casi una década por *Oracle*, con la intención de facilitar el desarrollo de programas sin tener en cuenta el hardware final ni el sistema operativo sobre el cual se ejecutaría. De esta manera, a partir de los programas desarrollados, se generaba un *código intermedio* que podía ejecutarse indistintamente en un sistema u otro que dispusiera del entorno de ejecución de Java (JRE). Todo ello facilitaba enormemente la portabilidad de programas reduciendo costes, minimizando el tiempo de implantación y eliminando la necesidad de desarrollar nuevas versiones de los programas para cada nueva plataforma.

Como ya se ha mencionado anteriormente, Java puede ser empleado para el desarrollo de aplicaciones de muy diverso tipo. Debido a ello hay una gran variedad de distribuciones disponibles, las cuales han ido evolucionando para adaptarse a las nuevas necesidades:

- ✓ *J2ME (Java 2 Micro Edition)* es la distribución destinada al desarrollo de software para dispositivos de recursos más limitados (electrodomésticos, TDT, impresoras, teléfonos móviles, PDA…). Tuvo mucho éxito debido al auge de los juegos en teléfonos móviles, aunque su uso ha decrecido notablemente.
- ✓ *J2SE (Java 2 Standard Edition)* es la distribución destinada al desarrollo de aplicaciones de escritorio para ordenadores personales o servidores, así como al desarrollo de aplicaciones cliente-servidor y conexiones con bases de datos (JDBC). Aunque no engloba completamente a J2ME, sí cuenta con una gran parte de las librerías en él contenidas.
- ✓ *J2EE (Java 2 Enterprise Edition)* es la distribución más completa de las tres y expande la funcionalidad de J2SE con soporte y API para el desarrollo de aplicaciones basadas en tecnologías web (*JSP*, *Servlets*, *WebServices*…), invocación de métodos remotos (*CORBA*, *RMI*)…

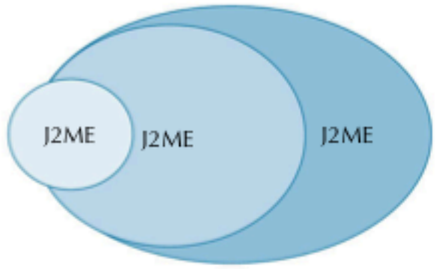


Figura 3.1
Distribuciones Java.

Actividades propuestas

3.1. Clasifica la tecnología Java en las diferentes categorías expuestas en el apartado 3.2. Lenguajes de programación.

3.2. Uno de los puntos fuertes de Java es que se trata de una tecnología que se puede utilizar de forma gratuita, pero no siempre siguió fielmente la filosofía *open source*. Esta situación fue bautizada como "la trampa de Java".

Lee el artículo "Libre pero encadenado. La trampa de Java", de Richard Stallman, publicado en GNU.org (accesible a partir del código QR adjunto) y busca información sobre cuál fue la situación inicial de Java con respecto al software libre y cuáles fueron los argumentos de peso que hicieron que la máquina virtual de Java no viniese incluida por defecto en las distribuciones GNU Linux.

3.3.2. Entorno de desarrollo JDK

Para poder desarrollar aplicaciones basadas en Java será necesario utilizar uno de los entornos de desarrollo suministrados por *Oracle*. Este entorno de desarrollo ofrecerá las herramientas necesarias para construir y ejecutar aplicaciones de casi cualquier tipo.

El entorno de desarrollo a emplear será el JDK *(Java Development Kit)*. Es recomendable elegir la última versión estable disponible, pero es probable que dicha versión presente algún tipo de incompatibilidad con herramientas que se necesiten instalar posteriormente (IDE).

Es importante destacar que existen dos tipos de JDK suministrados por *Oracle*, que, aun siendo similares y ofreciendo la misma funcionalidad, tienen distinta licencia:

- El primero de ellos es el JDK suministrado bajo licencia *Binary Code License Agreement for the Java SE Platform Products and JavaFX*.
- El segundo es el *OpenJDK* que es suministrado bajo licencia *GNU General Public License, version 2*.

Para ejecutar las aplicaciones desarrolladas con el JDK, independientemente de la variante usada, también será necesario el JRE *(Java Runtime Environment)*. Este vendrá suministrado directamente junto con el JDK.

Para saber más

La primera versión del *Java Development Kit (JDK 1.0)* fue liberada por Sun Microsystems el 23 de enero de 1996 con el lema "Escribe una vez, ejecuta en todas partes".

Desde Oracle se pretende lanzar cada tres años versiones "Soporte a largo plazo" (LTS), las cuales comenzaron con Java SE 11 en septiembre de 2018. En el siguiente enlace https://www.oracle.com/technetwork/java/java-se-support-roadmap.html podrás conocer en detalle la nueva filosofía de Oracle para el lanzamiento de versiones.

3.3.3. Entorno de ejecución JRE

Los lenguajes de programación de alto nivel ofrecen respecto a los lenguajes de bajo nivel una mayor abstracción sobre la arquitectura física a la hora de escribir programas. Para que se puedan obtener programas a partir de ellos es imprescindible la utilización de compiladores.

Originalmente los compiladores llevaban a cabo la traducción de los programas escritos, en un lenguaje de programación cercano al ser humano, a un código escrito en un lenguaje entendible por el sistema operativo sobre el cual se pretendía ejecutar. Algunos de los lenguajes que trabajan de esta manera son C, Pascal o Delphi.

Sin embargo, otras tecnologías muy relevantes en el mundo del desarrollo utilizan compiladores que traducen esos programas a un código intermedio que, por sí mismo, no puede ser ejecutado directamente por el sistema operativo. Ese código intermedio debe ser manejado por un elemento especial que se ejecuta sobre el sistema operativo y cuya función es traducir el código intermedio a instrucciones nativas entendibles por el sistema operativo.

En el mundo Java ese código intermedio se denomina *bytecode* y el elemento que hace de intermediario para que pueda ser ejecutado se denomina JRE *(Java Runtime Environment)*. Este está formado por una JVM *(Java Virtual Machine)*, bibliotecas con código Java y algunos componentes extra. Existen versiones para la mayor parte de los sistemas operativos del mercado, lo cual garantiza que el mismo código pueda ser ejecutado en diferentes sistemas sin necesidad de hacer modificaciones.

De forma análoga en el mundo .NET el "código intermedio" se denomina *CIL* o *MSIL* y la máquina virtual que lo traduce CLR *(Common Language Runtime)*.

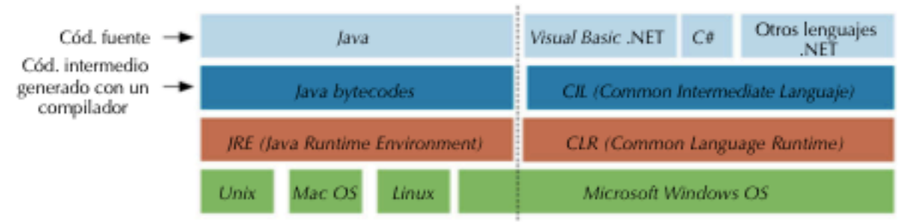


Figura 3.2
Java versus .NET.

Dado que la ejecución del código intermedio implicaría su traducción una y otra vez a lo largo de la vida del programa y esto ralentizaría en gran medida el tiempo de ejecución, se decidió incluir en el JRE un compilador JIT *(Just In Time)* que mejora notablemente el rendimiento de las aplicaciones y minimiza este retardo. Este compilador hace uso del código compilado con anterioridad para evitar interpretarlo nuevamente, además de otras acciones llevadas a cabo en pro de la mejora del rendimiento.

Sabías que...

Aunque Java fue diseñado para ejecutar sus programas sobre una máquina virtual, se han desarrollado compiladores que permiten generar ejecutables nativos que se pueden lanzar sobre el sistema operativo sin necesidad del JRE. Un ejemplo es GCJ (*GNU Compiler for Java*).

3.3.4. Variables de entorno PATH y CLASSPATH

A la hora de trabajar con Java es importante tener presente que hay dos variables de entorno cuyos valores tienen que estar correctamente configurados para permitir llevar a cabo la compilación, ejecución y generación de paquetes *jar* de una forma mucho más cómoda:

- *PATH:* es la variable del sistema utilizada por el sistema operativo para buscar los ejecutables de las diferentes aplicaciones desde el intérprete de comandos de MS-DOS o el terminal de Linux.
- *CLASSPATH:* es la variable del sistema utilizada por el entorno de ejecución de Java (JRE) para buscar los paquetes y clases definidas por el usuario.

Si se desea trabajar desde la línea de comandos o terminal, es de vital importancia configurar correctamente ambas variables para facilitar el trabajo.

Actividad propuesta 3.3

Accede a la página de descargas de *Oracle*, descarga la versión para su sistema operativo del *JDK 8* e instálalo. Por último, lleva a cabo la configuración de las variables de entorno *PATH* y *CLASSPATH*.

Recurso digital www

En los anexos web 3.2 y 3.3, disponibles en www.sintesis.com y accesibles con el código indicado en la primera página del libro, encontrarás información sobre la instalación y configuración el entorno.

3.3.5. Tipos de archivos empleados

Antes de comenzar a programar en Java, es necesario conocer los diferentes tipos de archivos que se van a manejar, así como las extensiones asociadas a estos:

- Cuando se escribe *código fuente* en el lenguaje de programación Java, este debe almacenarse en un archivo de texto plano y con extensión *.java*.
- Como resultado de compilar exitosamente un programa escrito en un archivo *.java* obtendremos, como mínimo, un archivo contenedor de código intermedio (*bytecode*) con extensión *.class*.
- A diferencia de otras tecnologías en las que el resultado de una correcta compilación es un archivo ejecutable, Java genera un archivo independiente por cada clase del proyecto con extensión *.class*. Esto puede hacer muy incómodo el manejo de los programas en aquellos proyectos que contengan un gran número de clases. Por este motivo Java ofrece la posibilidad de empaquetar ese conjunto de archivos *.class* y los recursos que estos utilizan (imágenes, archivos de texto plano…) en un único archivo con extensión *.jar* (*Java Archive*). Para la construcción de este tipo de archivos se emplea el algoritmo de compresión *Zip*.
- Siguiendo con la misma lógica y para facilitar la implantación de las aplicaciones en los servidores de aplicaciones, Java ofrece los archivos:

  a) *.war (Web Application Archive):* destinados a contener todos los archivos que forman una aplicación web desarrollada con tecnología Java (*JSP*, *Servlets*…) y facilitar su despliegue.
  b) *.ear (Enterprise Archive):* empaquetan uno o varios módulos en un archivo que facilita su despliegue en un único paso sobre un servidor de aplicaciones conforme a la estructura definida en el archivo contenido *META-INF/application.xml*. Puede, por tanto, contener archivos de cualquier tipo (incluidos *.jar* y *.war*).

3.3.6. Recolector de basura

Otro de los elementos importantes y diferenciadores de la tecnología Java es su recolector de basura (*garbage collector*). Este sigue una filosofía opuesta a la que impone la gestión manual de memoria, de manera que exime al programador de las tareas inherentes a su reserva y liberación.

Aquellos programadores provenientes de tecnologías como C o Delphi pueden ver esta característica como una ventaja o como un inconveniente:

- Ventaja porque facilita el desarrollo y evita errores humanos a la hora de gestionar la memoria. Gracias a esta liberación, es posible escribir código con la tranquilidad de que los programas desarrollados no terminarán consumiendo los recursos del sistema antes de que el programa finalice.
- Inconveniente si tenemos en cuenta que este elemento no supone un coste cero y afectará al rendimiento del programa, ya que consume recursos. Este es el motivo principal por el que se hace impensable el desarrollo de sistemas de tiempo real utilizando este tipo de tecnologías. Existen diferentes políticas de manejo del recolector de basura:

  a) Esperar a que no quede memoria libre y en ese momento ejecutar el recolector de basura.
  b) Fijar un umbral de ocupación de la memoria libre y, una vez alcanzado, ejecutar el recolector de basura.
  c) Ejecutar el recolector de basura a intervalos regulares.
  d) Ejecutar el recolector de basura justo antes de cada reserva de memoria.
  e) Permitir al programador que invoque explícitamente al recolector de basura cuando quiera.

Son varias las tecnologías que hacen uso del recolector de basura y cada una lo implementa de una forma particular. En el caso de Java, su funcionamiento se basa en la creación de una subrutina que se está ejecutando de forma paralela a los programas y que está controlando la memoria para detectar qué objetos están en uso y cuáles no. Su funcionamiento básico es el siguiente:

- *Marcado:* el recolector de basura identificará las zonas de memoria que ya no están en uso para su posterior liberación. Para ello identificará en el *Heap* las zonas de memoria de objetos que ya no tienen su correspondiente referencia en el *Stack*.
- *Borrado de objetos no referenciados:* el recolector de basura liberará las zonas de memoria en las que anteriormente había objetos que han quedado sin referencias que los apunten.
- *Compactación del espacio usado:* una vez liberado el espacio quedarán huecos en la memoria utilizada. El recolector de basura compactará el espacio usado para optimizar la ubicación en memoria de nuevos objetos que se vayan creando a lo largo de la ejecución.

Figura 3.3 Funcionamiento del recolector de basura (Java).

Recurso web

En el apartado 3.3 se ha explicado de forma muy básica cómo trabaja el recolector de basura de Java, pero este mecanismo es mucho más complejo. Para conocer con más precisión su funcionamiento, escanea el código adjunto.

3.4. Programación sin IDE

En los apartados 3.3.2 y 3.3.3 se han presentado las características de los entornos de desarrollo y ejecución para Java (JDK y JRE respectivamente), que suponen junto con un editor de texto plano los requisitos mínimos con los que afrontar el desarrollo de un proyecto. Estos podrán manejarse tanto desde un IDE con interfaz gráfica como desde línea de comandos.

El manejo de Java desde línea de comandos no entraña ninguna dificultad. Dado que los comandos serán equivalentes para cualquier sistema operativo, únicamente se mostrará su uso en uno de ellos y, por tanto, no se entrará en detalle acerca de cómo realizar las mismas tareas desde otro sistema. En el siguiente enlace podrás descargar los ficheros de ejemplo utilizados a lo largo de este apartado (pulsar para descargar).

3.4.1. Compilar un programa

Una vez se haya escrito el código fuente en un archivo de texto plano, se utilizará el comando *javac* para convertirlo a código intermedio. La sintaxis de uso del comando es la siguiente:

```
javac <opciones> <ficheros>
```

En el ejemplo mostrado en la figura 3.4 se pueden ver instrucciones equivalentes que generan el mismo resultado.

Figura 3.4
Compilación de programas desde línea de comandos.

En este caso se han creado cinco archivos con extensión *.class* como resultado del proceso de compilación. El nombre de cada archivo *.class* generado será el de la clase contenida dentro de cada archivo de código fuente compilado exitosamente. El código que contendrá deberá ser interpretado por la JRE.

Actividad propuesta 3.4

Ejecuta el comando *javac –help* desde un intérprete de comandos de MS-DOS o desde un terminal de Linux y estudia su sintaxis de uso.

3.4.2. Crear un jar

Una vez se hayan generado los archivos de código intermedio y se disponga de los recursos que estos utilizarán, se podrá generar un archivo *.jar* que los comprima para facilitar su transporte, almacenaje o inclusión en nuevos proyectos.

La sintaxis para emplear para el uso del comando *jar* es compleja, pues dispone de varias opciones:

```
jar {ctxui}[vfm0Me] [fichero jar] [fichero manifest] [punto de entrada a la
aplicación] [-C dir] ficheros ...
```

Estos archivos podrán construirse con las opciones *–cvf* para indicar que se quiere crear un archivo nuevo con un nombre determinado. Podrán construirse como:

- ✓ *Simples contenedores* de recursos que serán utilizados en otros proyectos; en este caso bastaría con generar el archivo *.jar*.

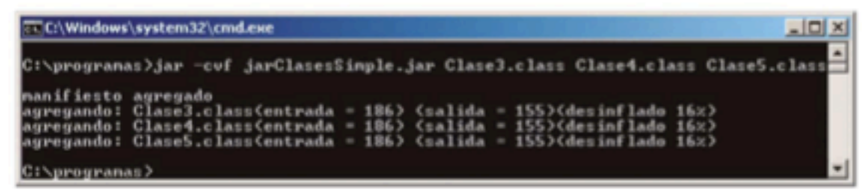


Figura 3.5
Creación de *jar* contenedor.

- ✓ *Ejecutables comprimidos*. En este caso sería necesario añadir la opción *–e* además de indicar qué clase será la principal.

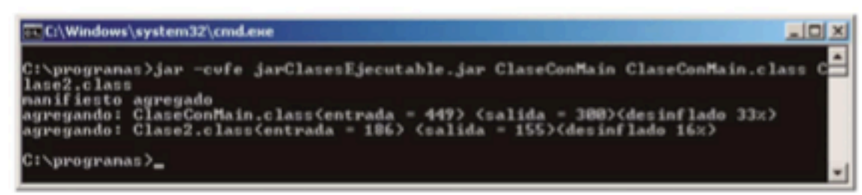


Figura 3.6
Creación de *jar* ejecutable.

Sobre cualquier archivo *.jar* se podrá añadir (opción *–u*) o consultar contenido (opción *–t*), aunque no existen opciones para eliminar elementos contenidos en él y esta tarea debe realizarse con otras herramientas (WinZip, WinRAR, 7zip…).

```
C:\Windows\system32\cmd.exe
C:\programas>jar -ft jarClasesEjecutable.jar
META-INF/
META-INF/MANIFEST.MF
ClaseConMain.class
Clase2.class

C:\programas>jar -uf jarClasesEjecutable.jar Clase3.class

C:\programas>jar -ft jarClasesEjecutable.jar
META-INF/
META-INF/MANIFEST.MF
ClaseConMain.class
Clase2.class
Clase3.class

C:\programas>
```

Figura 3.7
Consulta y modificación de contenidos de un *jar*.

Cuando estos archivos se construyen, dentro del archivo contenedor se creará un directorio *META-INF* en el que se podrá encontrar el archivo de texto plano *MANIFEST.MF*. Este archivo contiene cabeceras con información importante relativa al archivo *jar* (por ejemplo, la clase que se haya especificado principal en un ejecutable).

```
MANIFEST.MF: Bloc de notas
Archivo Edición Formato Ver Ayuda
Manifest-Version: 1.0
Created-By: 1.8.0_181 (Oracle Corporation)
Main-Class: ClaseConMain
```

Figura 3.8
Contenido del fichero MANIFEST.MF de un *jar*.

Actividad propuesta 3.5

Ejecuta el comando *jar* desde un intérprete de comandos de MS-DOS, o desde un terminal de Linux, y estudia su sintaxis de uso.

3.4.3. Ejecutar un programa

Una vez se haya generado el código intermedio y se disponga de un archivo *.class* o de un *.jar* ejecutable, podrá lanzarse sobre la JRE para que esta lo interprete y ejecute. La sintaxis de uso del comando es la siguiente:

```
java [opciones] <clase principal> [argumentos…]
```



```
C:\Windows\system32\cmd.exe
C:\programas>javac HolaMundo.java

C:\programas>java HolaMundo
Hola Mundo !

C:\programas>java -jar jarClasesEjecutable.jar
Ejecutando desde ClaseConMain !

C:\programas>
```

Figura 3.9
Ejecución de programas.

Recurso web

En la URL https://compiler.javatpoint.com/opr/test.jsp puedes encontrar una herramienta online que permite compilar y ejecutar programas sin necesidad de instalar ninguna aplicación.

Actividad propuesta 3.6

Ejecuta el comando *java –help* desde un intérprete de comandos de *MS-DOS* o desde un terminal de Linux y estudia el resultado.

Investiga

Averigua cómo ejecutar programas Java ubicados en el directorio *C:/programas* sobre un sistema operativo Microsoft Windows, o */programas* si trabaja en Linux, sin necesidad de crear la variable de entorno *CLASSPATH*.

3.5. Programación con IDE

A lo largo de los apartados anteriores se han descrito las características principales de los entornos de desarrollo y ejecución (JDK y JRE, respectivamente), mostrándose cómo hacer uso de estas herramientas desde línea de comandos.

Como seguramente se ha podido comprobar, llevar a cabo el desarrollo de proyectos de gran envergadura utilizando únicamente la línea de comandos puede ser una tarea ardua y propensa a cometer errores. Por este motivo surgieron los entornos de desarrollo integrado.

3.5.1. Características y elementos

Este tipo de aplicaciones, también conocidas como IDE *(Integrated Development Environment)*, son un conjunto de herramientas que han sido generadas para facilitar el desarrollo de programas y que se suministran de forma conjunta.

Generalmente este tipo de paquetes ofrecen un ambiente de desarrollo basado en aplicaciones gráficas desde las que es muy fácil acceder a todas las herramientas, la mayor parte de ellas no disponibles en un simple JDK.

Las herramientas que, de forma general, se pueden encontrar en los IDE son un editor, un ensamblador, un compilador, un enlazador, un depurador, asistentes que facilitan el desarrollo, un gestor de conexiones a diferentes orígenes de datos…

3.5.2. Alternativas

Existen varias alternativas en el mercado tanto de pago como libres. Algunas están orientadas al desarrollo en un único lenguaje de programación mientras que otras se adaptan al desarrollo en varias tecnologías.

Entre todas ellas hay que destacar *Eclipse* y *NetBeans*. Ambos pueden ser instalados en sistemas operativos Microsoft Windows y Linux (entre otros), se pueden descargar libremente, tienen soporte para desarrollar en varios lenguajes…

Otras alternativas son:

- *IntelliJ IDEA:* disponible para sistemas operativos Microsoft Windows y Linux. Puede incorporar soporte para varios lenguajes de programación. Está desarrollado en Java y dispone de versiones gratuitas y de pago.
- *JCreator:* únicamente hay versiones para el sistema operativo Microsoft Windows. Está desarrollado en *C++*, lo que lo hace muy recomendable para máquinas con escasos recursos. Tiene una versión gratuita y otra de pago.

Recurso digital www

En el anexo web 3.4, disponible en www.sintesis.com y accesible con el código indicado en la primera página del libro, encontrarás información sobre la instalación de *NetBeans*.

Actividad propuesta 3.7

Accede a la página de descargas de *NetBeans*, descarga la versión correspondiente a su sistema operativo e instálalo.

Investiga

Aunque este libro propone la instalación de *NetBeans* como IDE de trabajo, posiblemente consideres que otra de las alternativas propuestas se adecúa mejor a tus necesidades. Investiga y profundiza sobre las características de los IDE presentados, así como de otros que consideres oportuno, y debate en clase con tus compañeros sobre cuál es el más apropiado.

3.5.3. Configuración de NetBeans

Las configuraciones para realizar sobre el IDE pueden ser de tipo general, afectando a todos los proyectos, o de tipo específico, afectando únicamente a un proyecto. Aunque en el capítulo siguiente se explicará en detalle qué es un proyecto, en este momento se puede entender como un conjunto de recursos que representan al programa que se va a construir y que están organizados en una carpeta.

A) Configuraciones generales

Para hacer configuraciones generales hay que acceder a *Tools > Options*.

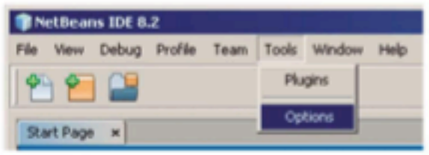


Figura 3.10
Acceso a opciones generales en *NetBeans*.

El primer paso será activar las características Java para que el entorno se quede preparado para trabajar con esta tecnología. Para ello activar el soporte para Java desde la opción Java (pestaña *GUI Builder* o *Java Debugger*).

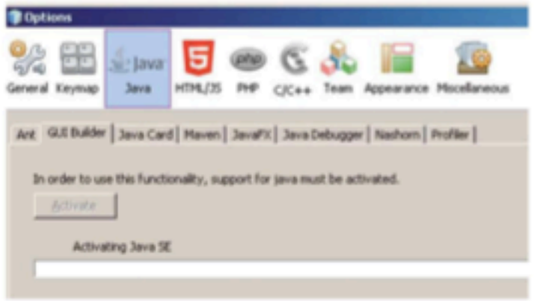


Figura 3.11
Activación de características.

Automáticamente se activarán todas las herramientas del IDE necesarias para el desarrollo de nuestros proyectos y se habrán habilitado también nuevas configuraciones generales hasta ahora ocultas.

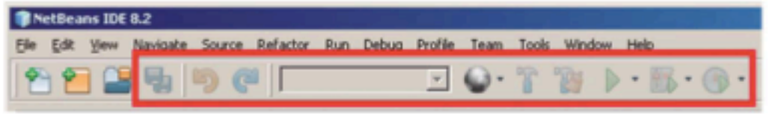


Figura 3.12
Nuevos accesos a herramientas.

Figura 3.13
Nuevas configuraciones habilitadas.

RECUERDA

✓ Las configuraciones establecidas por defecto generalmente son buenas para casi cualquier desarrollador y proyecto, por lo que es recomendable no modificarlas.

A continuación, se describen las configuraciones generales más interesantes:

- ✓ *Editor:* establecerá cómo realizar las tabulaciones en el código (*Formatting*), cómo trabajará el autocompletado (*Code Completion*), cómo se optimizará y corregirá el código (*On Save*), los tipos de mensajes de error o advertencia mostrados por el entorno (*Hints*)…
- ✓ *Fonts & Colors:* desde esta opción es posible configurar la tipografía y la simbología empleada por el editor.
- ✓ *Keymap:* usado principalmente para establecer *shortcuts*.
- ✓ *Java*: permite configurar el comportamiento del constructor (*GUI Builder*) y del depurador (*Java Debugger*).
- ✓ *Appearance:* desde esta opción será posible configurar el aspecto del IDE (separadores de documentos, ventanas y aspecto).

También es recomendable mostrar las ventanas *Projects* y *Files* (proyectos y ficheros, respectivamente) para tener acceso desde el IDE a la estructura del proyecto.

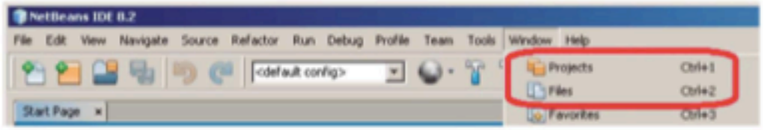


Figura 3.14
Acceso a ventanas *Projects* y *Files*.

Actividad propuesta 3.8

Configura el entorno de trabajo para que muestre las ventanas *Projects* y *Files*.

B) *Configuraciones específicas de cada proyecto*

Todos los proyectos recién creados atenderán a las configuraciones que se hayan establecido de forma general en el *IDE*. Sin embargo, es posible matizar estas configuraciones para realizar los cambios oportunos sobre algunas de ellas dejando intactas las demás.

Para acceder a las configuraciones específicas de un proyecto, será necesario hacer clic derecho sobre el proyecto y seleccionar la opción *Properties*. Aparecerá una nueva ventana desde la que podrán configurarse las propiedades del proyecto (algunas de ellas similares a las configuradas de forma general y otras totalmente específicas).

Dado que aún no se ha descrito cómo crear un proyecto, se recomienda retroceder a este apartado una vez se haya completado el apartado 3.5.4.

3.5.4. Manejo del IDE

A) *Crear un proyecto*

Una vez se ha realizado la instalación y configuración del IDE, ya se puede comenzar a escribir el primer programa. *NetBeans* está diseñado para trabajar por proyectos y, por tanto, aunque permite crear archivos independientes de algunas tecnologías como *HTML* o *JavaScript*, el primer programa deberá ser creado dentro de un proyecto.

Como se puede observar, *NetBeans* ofrece en su barra de tareas iconos para las acciones más habituales. Para crear el proyecto se debe hacer clic en el icono de la carpeta. También se puede hacer uso del atajo de teclado correspondiente.

Figura 3.15
Creación de proyectos.

Automáticamente se abrirá un asistente que guiará el proceso. Se debe seleccionar *Java Application* y, en la ventana siguiente, establecer el nombre y el directorio del proyecto. Será posible crear la clase principal automáticamente dentro del proyecto si se dejó el *checkbox Create Main Class* marcado.

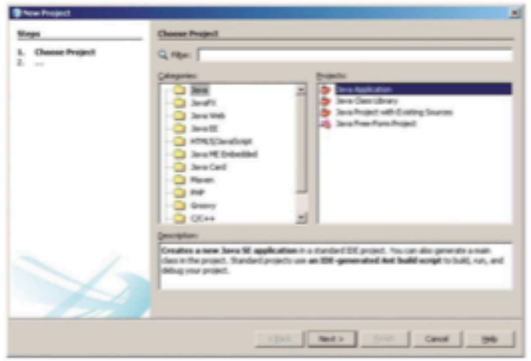


Figura 3.16
Selección del tipo de proyecto.

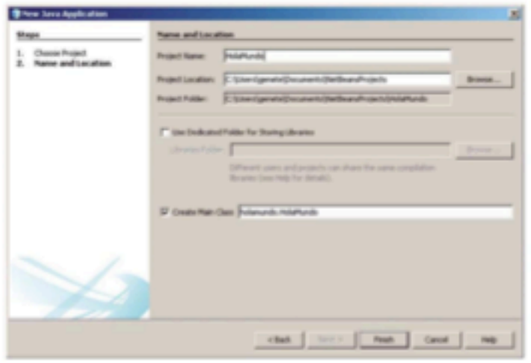


Figura 3.17
Establecimiento de las propiedades del proyecto.

Como resultado de este paso se habrá generado el proyecto en el directorio indicado y la estructura del programa que contendrá la clase principal.

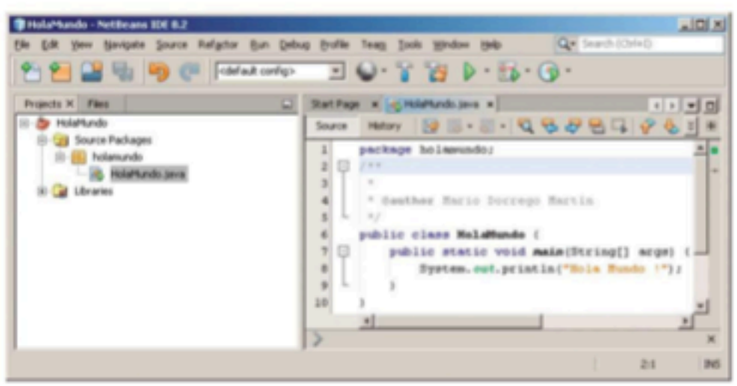


Figura 3.18
Proyecto HolaMundo.

B) Construir el proyecto

Una vez se haya generado el proyecto y escrito el código del programa, se deberá generar el código intermedio que será almacenado en los archivos *.class*. Para ello el entorno ofrece dos alternativas:

- *Build Project:* construye el proyecto, compilando únicamente los archivos de código fuente que sea necesario y modificando el *jar* con dicho contenido.
- *Clean and Build Project:* elimina todo resultado de la construcción anterior, compila todos los archivos de código fuente, enlaza los contenidos y construye un archivo *jar* que contendrá los archivos compilados y recursos que utilicen.

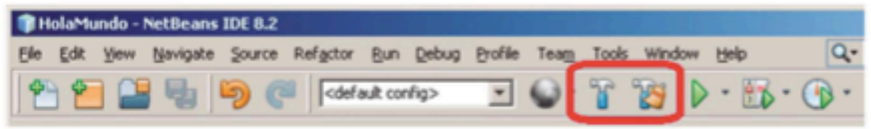


Figura 3.19
Herramientas para la construcción de proyectos.

La primera vez que se construya el proyecto ambas alternativas provocarán el mismo resultado. Desde la ventana *Files* se podrá tener acceso a la estructura de directorios generada para el proyecto.

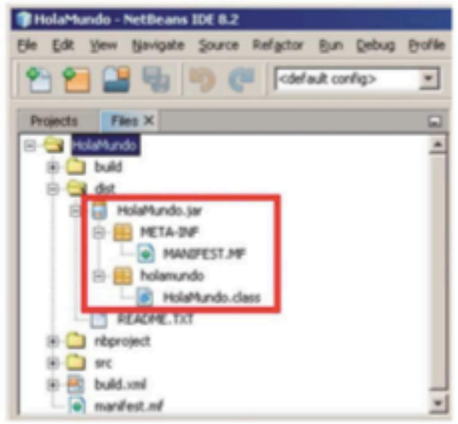


Figura 3.20
Estructura de directorios del proyecto.

C) Ejecutar un programa

El proyecto libre de errores habrá dado como resultado un código intermedio que estará listo para ser ejecutado. Para ejecutar el código y visualizar el resultado existe la opción *Run Project*. El resultado del programa podrá observarse en la ventana *Output*.

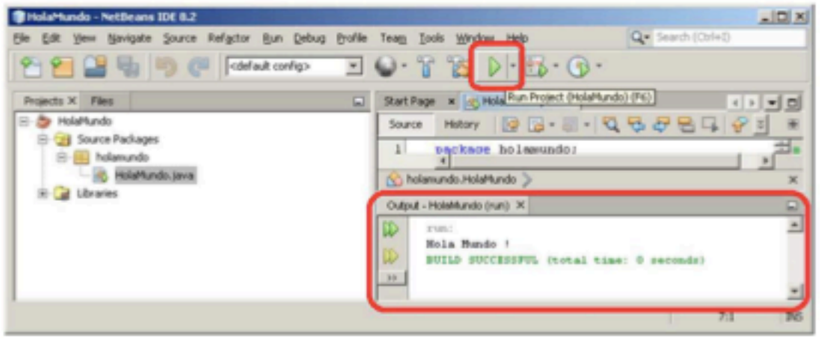


Figura 3.21
Ejecución de proyectos.

D) Depurar un programa

De la misma forma que se puede ejecutar el código, también se podrá depurar para corregir posibles errores producidos en tiempo de ejecución o estudiar su comportamiento. En la opción *Debug* del menú de opciones se encuentran los ítems de acceso a las diferentes opciones que permiten realizar esta tarea. Entre estas se encuentran:

- *Ctrl + F5*: comenzar el *Debug*.
- *Mayús + F5*: detener el *Debug*.
- *F5*: continuar con la ejecución hasta el siguiente punto de ruptura o hasta el final del programa.
- *F4*: ejecutar el programa hasta la posición actual del cursor.
- *F7*: ejecutar la siguiente línea del programa. Si la siguiente instrucción es la llamada a un método, continuará con la ejecución línea a línea dentro del método.
- *F8*: ejecutar la siguiente línea del programa. Si la siguiente instrucción es la llamada a un método, ejecutará el método como si de una instrucción simple se tratase.
- *Ctrl + F8*: establecer o eliminar un punto de ruptura sobre la línea actual del cursor.
- *Ctrl + Mayús + F7*: añadir una nueva ventana de inspección (*Watch*).

Otra vía de acceso a las mismas acciones es la barra de tareas con iconos (figura 3.22).

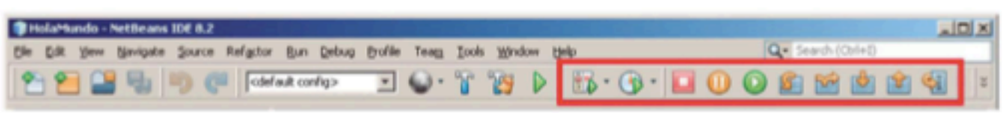


Figura 3.22
Herramientas para la depuración de programas.

Al igual que en la ejecución, los resultados de la depuración podrán observarse en la ventana *Output*, mientras que el valor de las variables manejadas en el programa podrá visualizarse en la ventana *Variables*.

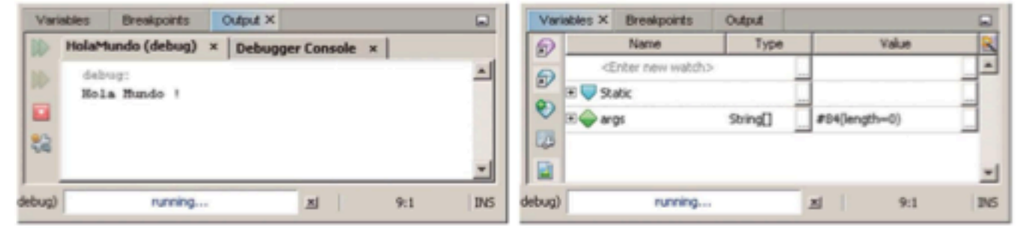


Figura 3.23
Ventanas resultado de la depuración e inspección de variables.

También será posible consultar el valor de ciertas variables situando el cursor del ratón sobre ellas cuando el programa haya quedado detenido en algún punto de ruptura.

Resumen

- Hoy en día existe una gran variedad de lenguajes y tecnologías con los que afrontar el desarrollo de un proyecto.
- Los lenguajes de programación pueden clasificarse según varios aspectos.
- Algunos lenguajes no se ejecutan directamente sobre el sistema operativo, sino que requieren de un elemento intermedio o máquina virtual.
- Java es uno de los lenguajes más utilizados en la actualidad y dispone de recursos para el desarrollo de aplicaciones de casi cualquier tipo.
- Para trabajar en Java es necesario el JDK y el JRE, así como unas configuraciones mínimas del sistema.
- La utilización de un IDE facilita el desarrollo de programas y minimiza los errores.

Ejercicios propuestos

1. Aunque la primera versión del JDK fue liberada en 1996, los orígenes de Java se remontan a un periodo anterior. Investiga cuál fue el punto de partida del proyecto e indica los diferentes nombres que fue tomando hasta bautizarse como Java.
2. Ordena los lenguajes de programación que han sido citados en el capítulo por orden cronológico.
3. ¿Utilizarías Java para el desarrollo de un sistema de tiempo real? Justifica tu respuesta.
4. Describe las diferencias existentes entre la arquitectura Java y .NET.
5. Investiga las diferencias existentes en la forma de trabajar de un compilador y un intérprete de código.
6. Enumera y describe tres características de Java que hacen que se trate de una buena alternativa a la hora de afrontar un proyecto.
7. Averigua cómo reestablecer la disposición de las ventanas del IDE *NetBeans* a su posición por defecto.

Supuestos prácticos

1. Sobre el entorno de desarrollo *NetBeans* recién instalado, crea tu primer proyecto "Hola Mundo", construye el *jar* y ejecútalo.
2. A partir del proyecto creado en el supuesto práctico anterior, añade un punto de ruptura y depura el proyecto para conseguir que la ejecución se detenga en dicho punto. ¿Has podido establecer el punto de ruptura en cualquier línea?
3. Ejecuta el programa paso a paso hasta que finalice. ¿Qué comando de depuración has utilizado? ¿Podrías haber utilizado otro obteniendo el mismo resultado?
4. Repite la depuración del programa y, una vez quede detenido en el punto de ruptura, finaliza la ejecución (salida forzosa).
5. Realiza la instalación y configuración de un nuevo entorno de desarrollo integrado y repite los apartados anteriores.

Actividades de autoevaluación

1. Si se necesita desarrollar una aplicación de escritorio para un ordenador personal:
   - ☐ *a)* Podré utilizar *J2ME, J2SE* o *J2EE* indistintamente.
   - ☐ *b)* Podré utilizar *J2SE* o *J2EE*, pero no *J2ME*.
   - ☐ *c)* Podré utilizar *J2ME* o *J2EE*, pero no *J2SE*.
   - ☐ *d)* Podré utilizar *J2ME* o *J2SE*, pero no *J2EE*.

2. Para el desarrollo de aplicaciones Java sin IDE en Windows será estrictamente necesario configurar como mínimo:
   - ☐ *a)* La variable de entorno *PATH*.
   - ☐ *b)* Las variables de entorno *JAVA_HOME, PATH* y *CLASSPATH*.
   - ☐ *c)* Las variables de entorno *PATH* y *CLASSPATH*.
   - ☐ *d)* Todas las opciones son incorrectas (se puede trabajar sin configurar variables de entorno).

3. ¿Cuál de los siguientes IDE únicamente está disponible para sistemas operativos Windows?
   - ☐ *a)* *Eclipse*.
   - ☐ *b)* *Netbeans*.
   - ☐ *c)* *JCreator*.
   - ☐ *d)* *IntelliJ IDEA*.

4. ¿Cómo se puede conocer qué clase contiene el programa principal de un *jar* ejecutable?
   - ☐ *a)* Consultando si el archivo *MANIFEST.MF* contenido en *META-INF* declara la cabecera "*Main-Class: ...*".
   - ☐ *b)* Consultando si el archivo *META-INF* declara la cabecera "*Main-Class:...*".
   - ☐ *c)* Una vez creado el *jar* no es posible conocer esa información.
   - ☐ *d)* Todas las opciones son incorrectas.

5. El código intermedio Java se denomina:
   - ☐ *a)* *Bitecode*.
   - ☐ *b)* *Bytecode*.
   - ☐ *c)* *CIL*.
   - ☐ *d)* *CLR*.

6. ¿Cuál es la extensión del fichero que contiene los ficheros de una aplicación web Java?
   - ☐ *a)* *.class*
   - ☐ *b)* *.jar*
   - ☐ *c)* *.war*
   - ☐ *d)* *.ear*

7. El atajo que permite ejecutar el programa hasta la posición actual del cursor es:
   - ☐ *a)* F4.
   - ☐ *b)* Ctrl + F5.
   - ☐ *c)* F5.
   - ☐ *d)* F7.

8. El orden en que se ejecutan las fases del recolector de basura es:
   - ☐ *a)* Borrado, marcado y compactación.
   - ☐ *b)* Marcado, borrado y compactación.
   - ☐ *c)* Marcado, compactación y borrado.
   - ☐ *d)* El orden no es relevante, por lo que cualquiera de las opciones anteriores es correcta.

9. ¿Cuál de las siguientes instrucciones permite ejecutar *programa.jar*?
   - ☐ *a)* *Jar programa.jar*
   - ☐ *b)* *Java -jar programa*
   - ☐ *c)* *Java -jar programa.class*
   - ☐ *d)* *Java -jar programa.jar*

10. Identifica la afirmación falsa:
    - ☐ *a)* *F8* es el atajo que, en modo depuración, permite ejecutar la siguiente línea del programa. Si la siguiente instrucción es la llamada a un método, ejecutará el método como si de una instrucción simple se tratase.
    - ☐ *b)* El compilador *Just In Time (JIT)* mejora notablemente el rendimiento de las aplicaciones al hacer uso del código compilado con anterioridad para evitar interpretarlo nuevamente.
    - ☐ *c)* Existen algunas líneas de código sobre las cuales no es posible poner un punto de ruptura para realizar la depuración.
    - ☐ *d)* La variable del sistema *PATH* es utilizada por el entorno de ejecución de Java (JRE) para buscar los paquetes y clases definidas por el usuario.

SOLUCIONES:

1.	a	**b**	c	d	5.	a	**b**	c	d						
2.	a	b	c	**d**	6.	a	b	**c**	d	9.	a	b	c	**d**	
3.	a	b	**c**	d	7.	**a**	b	c	d	10.	a	b	c	**d**	
4.	**a**	b	c	d	8.	a	**b**	c	d						

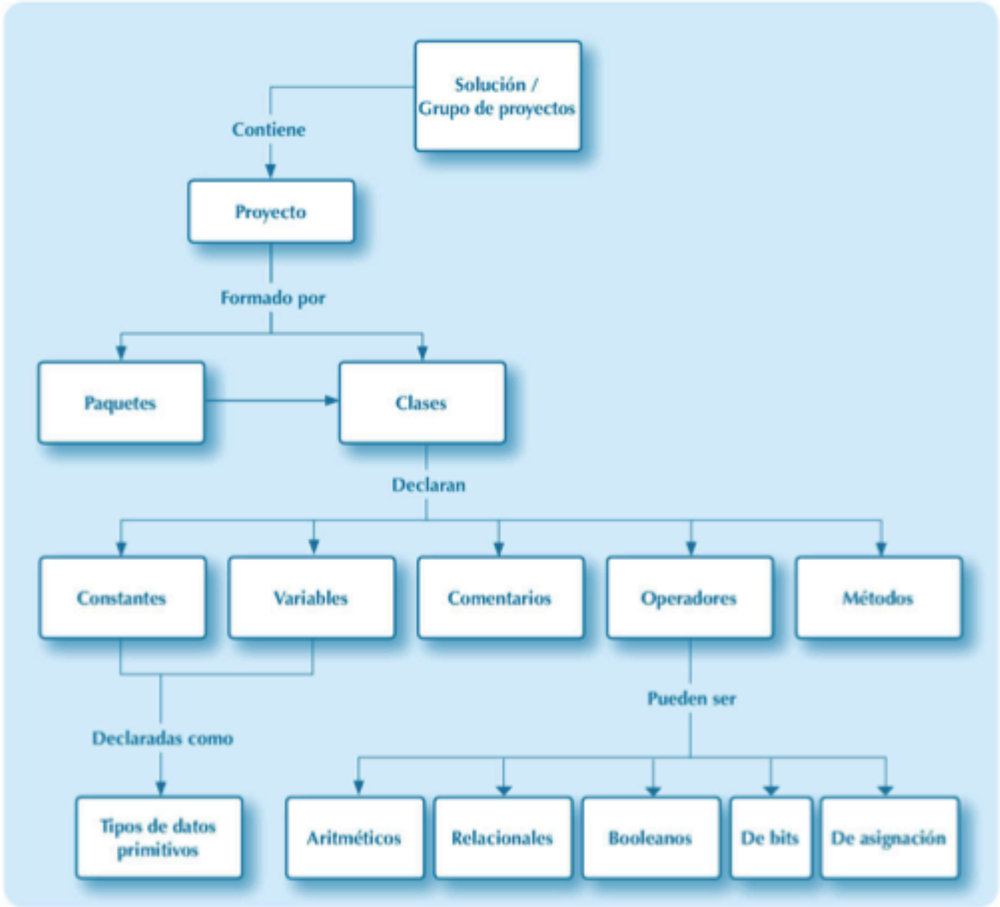
4

Elementos de un programa informático

Objetivos

- ✓ Analizar la estructura y elementos básicos de un programa informático.
- ✓ Conocer los tipos de datos primitivos.
- ✓ Comprender el uso de operadores y conocer su orden de precedencia.
- ✓ Aprender a identificar correctamente los diferentes elementos de un programa.

Mapa conceptual



Glosario

lowerCamelCase. Convención que propone que todas las palabras vayan juntas y la primera letra de cada una de las ellas, excepto en la primera palabra, sea mayúscula.

lowercase. Convención que propone escribir todos los identificadores en minúsculas.

Main. Conocido también como programa principal, es la parte de un programa que se corresponde con el punto de entrada a este y que recoge los argumentos principales con los que es invocado. En programación orientada a objetos (lenguaje Java o C++) es un método contenido dentro de una clase denominada clase principal. En programación estructurada (lenguaje C) se corresponde con una función.

Memoria principal. Dispositivo en el que se almacena de forma temporal la información necesaria para la ejecución del sistema operativo y los programas que corren sobre él.

Notación. Sistema de signos destinado a representar conceptos de un campo determinado (música, matemáticas…).

Scope. También conocido como *ámbito* o *alcance,* es la sección de código de un programa en la que una variable es accesible.

SCREAMING_SNAKE_CASE. Convención que propone que todas las palabras vayan en mayúsculas y separadas por guion bajo "_".

Secuencia de escape. Combinación de caracteres interpretada como caracteres no imprimibles o acciones a llevar a cabo.

Terminal. Originariamente el término terminal se utilizaba para definir a los diferentes dispositivos que podían manejar una computadora a distancia. Hoy en día este término se utiliza para definir la interfaz en línea de comandos que los diferentes sistemas operativos ofrecen para su manejo, aunque sería más propio en este caso hablar de pseudoterminal o consola.

UpperCamelCase. Convención que propone que todas las palabras vayan juntas y la primera letra de cada una de ellas sea mayúscula.

4.1. Soluciones y proyectos

En este capítulo se realizará un estudio de los diferentes elementos que conforman un programa informático desarrollado en Java. A lo largo de este capítulo se irán abordando todos ellos desde un punto de vista más genérico hacia uno más específico.

El primero de los elementos que tratar es el *proyecto*. Aunque este concepto ya apareció en el capítulo anterior cuando se realizaba la instalación del IDE y se explicaban las características de este, aún no se ha profundizado en qué consiste y cuál es su función. Se entiende como *proyecto* a un conjunto de carpetas y archivos organizados conforme a cierto criterio que permita llevar un orden lógico. En la tecnología Java un proyecto suele constar de archivos de código fuente (*.java*), archivos de código intermedio (*.class*) y otros archivos que son empleados desde el programa (imágenes, archivos de texto plano…). El primer paso para desarrollar un programa será crear el proyecto.

Una *solución* es un conjunto de uno o más proyectos relacionados entre sí, aunque también puede contener archivos que no tengan conexión directa con ninguno de los proyectos de la solución. Este concepto es empleado comúnmente en la tecnología .NET mientras que en *NetBeans* se utilizan los *Project Groups* (grupos de proyectos).

Figura 4.1
Grupo de proyectos.

4.2. Identificadores

Un *identificador* es una secuencia de uno o más caracteres asignada por el programador a un elemento del programa (constante, variable, método, clase, paquete…). Todo identificador debe estar formado conforme a las siguientes reglas:

- Cada identificador debe contener, al menos, un carácter.
- Los caracteres válidos serán letras (a-Z), dígitos (0-9), guion bajo (_) o dólar ($).
- El primer carácter no podrá ser un dígito.

Aunque el lenguaje es muy flexible respecto a los identificadores, es conveniente seguir algunas recomendaciones:

- ✓ Para *paquetes* utilizar *lowercase* sin espacios entre palabras. Puesto que un paquete podrá contener a otro, la jerarquía de paquetes se representará con el carácter punto ".".
- ✓ Para *clases* e *interfaces* utilizar *UpperCamelCase*.
- ✓ Para *variables* y *métodos* utilizar *lowerCamelCase*.
- ✓ Para *constantes* utilizar *SCREAMING_SNAKE_CASE*.

Actividad propuesta 4.1

Además de las convenciones anteriormente citadas existen otras que en algún momento puede ser de utilidad conocer y aplicar. Busca información sobre las convenciones *notación húngara, leet speak, snake_case, kebab-case* y *Train-Case,* y propón un ejemplo de aplicación de cada una de ellas.

4.3. Palabras reservadas

Las palabras reservadas son, tal como su propio nombre indica, un conjunto de identificadores que no pueden ser utilizados en los programas puesto que tienen su propio significado. En el cuadro 4.1, se muestra un listado de estas palabras.

Cuadro 4.1
Palabras reservadas

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

4.4. Clases

Entre los objetivos de la programación orientada a objetos se encuentran la modularidad y la encapsulación (principios heredados del paradigma de programación modular). Para conseguir alcanzar estos objetivos la tecnología Java propone, entre otros, el elemento *Clase*.

El concepto genérico de clase ya fue introducido en el capítulo 2 (Programación orientada a objetos) y no será hasta el capítulo 6 (Clases y utilización de objetos), donde se explicará en detalle en qué consiste y cómo se maneja en la tecnología Java. En el siguiente apartado puede verse un ejemplo sencillo de declaración de clase.

4.4.1. Método *main*

Todos los programas tienen un punto de entrada a partir del cual podrán comenzar la ejecución. En el caso de los programas Java este punto de partida es el método *main*. Su declaración debe hacerse en alguna de las clases del programa y debe ser inalterable en cuanto a que siempre debe ser público, estático y mantener la especificación de los parámetros de entrada. Además, el identificador de la clase en la que se declare debe coincidir plenamente con el nombre del archivo en el cual se ha escrito. A continuación, puede verse un ejemplo de declaración de clase con un programa principal:

```
public class HolaMundo {
   public static void main(String[] args) {
        System.out.println("Hola Mundo !");
 }
}
```


4.5. Paquetes

Otro de los elementos que Java propone para conseguir modularidad y encapsulación es el *paquete*. El paquete es un mecanismo que ofrece varias ventajas en el desarrollo:

- Agrupa clases con características comunes.
- Independiza declaraciones realizadas en un paquete de las existentes en otros (lo que permite utilizar los mismos identificadores si fuese necesario).
- Proporciona un mecanismo de control de acceso mediante la configuración de la visibilidad a nivel de paquete, la cual afecta a todos los elementos en él contenidos.

Los paquetes se declaran utilizando la palabra reservada *package* seguida del nombre del paquete en la primera línea del código del archivo y finalizando con el carácter punto y coma (;). El nombre del paquete puede estar formado por varias palabras que irán en todo caso seguidas y sin separadores. Para cada clase que vaya a pertenecer al paquete es necesario incluir dicha declaración en la primera línea del archivo en el que esté declarada.

Los paquetes pueden contener a su vez otros paquetes en lo que se conoce como *jerarquía de paquetes*. Para representar dicha jerarquía los nombres de los paquetes deben especificarse separándose por puntos. Para acceder a los contenidos de un paquete desde clases pertenecientes a él bastará con indicar el nombre de la clase. Sin embargo, si las clases pertenecen a otro paquete distinto, se deberá anteponer el nombre del paquete al de la clase o bien realizar la importación de los contenidos del paquete. Las importaciones de paquetes se realizarán después de las declaraciones de pertenencia al paquete de la clase actual y se podrá utilizar el carácter asterisco (*) para referirse a todos los elementos en él contenidos:

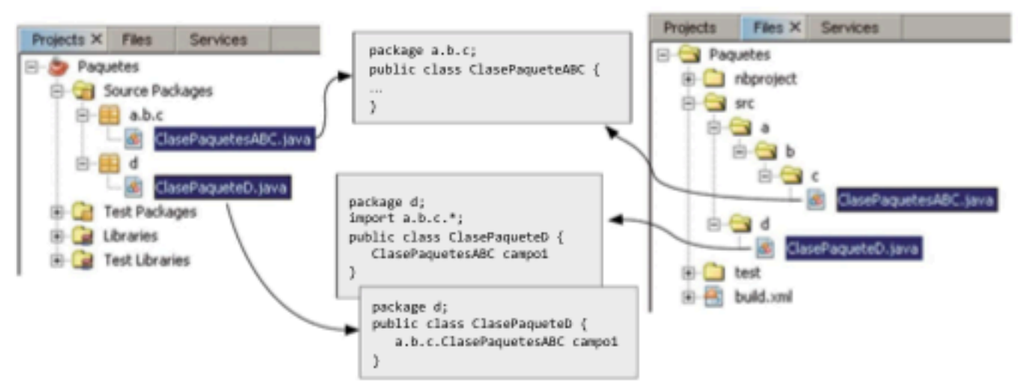


Figura 4.2
Estructura lógica y física con paquetes.

A la hora de trabajar con paquetes desde el terminal, es importante tener en cuenta que, para compilar un programa, conviene indicar el directorio dentro del cual depositar los ficheros *.class* (modificador *–d*) y que, para la ejecución simple o la creación del *jar* ejecutable, el nombre de la clase que implementa el método *main* deberá ir precedido por el nombre de los paquetes que la contiene:

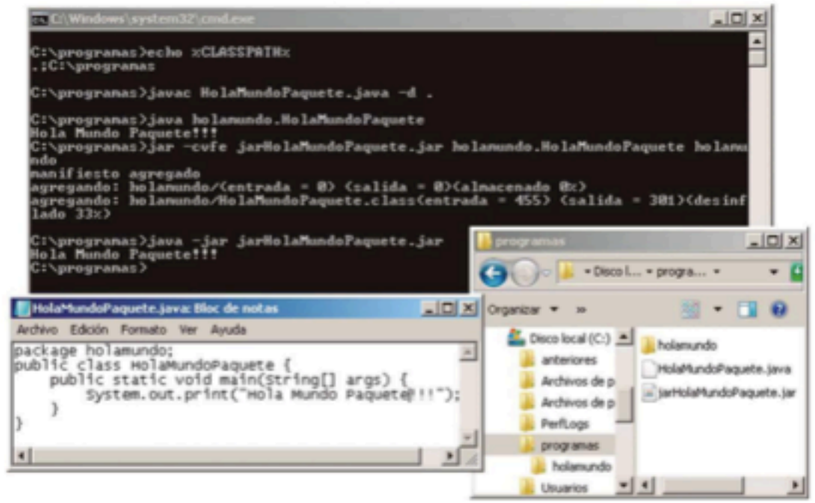


Figura 4.3
Manejo de comandos con paquetes.

4.6. Variables

Las variables son uno de los elementos clave en programación, ya que permiten almacenar en *memoria principal* diferentes valores a lo largo de la vida del programa. En Java pueden distinguirse dos tipos bien diferenciados de variables: las de tipo primitivo y las referencias a objetos (en el apartado 4.8 se describirán los tipos de datos). La declaración e inicialización puede hacerse en la misma línea o bien en líneas independientes. En cualquier caso, cada línea deberá ir finalizada por ";". A continuación, se muestra un ejemplo de la sintaxis empleada para declarar variables (las palabras entre corchetes son opcionales):

```
[Visibilidad] tipo nombreVariable1;
nombreVariable1 = valor1;
[Visibilidad] tipo nombreVariable2 = valor2;
```

4.6.1. Ámbito de una variable

Una variable será accesible desde una sección de código u otra dependiendo de dónde y cómo se declare. A esto se le conoce como ámbito, alcance o *scope* de la variable. Existen algunas consideraciones generales a tener en cuenta sobre el ámbito de una variable:

- ✓ El código contenido entre un par de llaves "{}" se encuentra dentro del mismo ámbito.
- ✓ Una variable declarada en un ámbito puede ser accedida desde el código de ese ámbito y desde el código de los subámbitos que sean creados dentro de este.
- ✓ Una vez se salga del ámbito en el que fue declarada la variable, esta será destruida.

A continuación, se muestra un código de ejemplo en el que se puede observar cómo afecta el ámbito de declaración de una variable a la hora de ser accedida:

Como se puede comprobar, en la sintaxis de declaración de una variable se incluye un elemento que permite establecer la visibilidad de esta. En el caso de que se trate de una variable miembro (campo), el modificador de visibilidad no afectará al ámbito de dicha variable dentro de la clase. Esta y otras circunstancias especiales que afectan al ámbito de las variables serán tratadas con posterioridad en los siguientes capítulos.

4.7. Constantes

Una constante es un elemento que mantiene un valor inalterable a lo largo de toda la vida del programa. En Java se declaran como una simple variable, pero anteponiendo *static final* al tipo. La declaración e inicialización debe hacerse en la misma línea. A continuación, se muestra la sintaxis que debe emplearse:

```
[Visibilidad] static final tipo NOMBRE_CONSTANTE = valor;
```

4.8. Tipos de datos primitivos

Los tipos de datos delimitan claramente qué tipo de valores podrán ser contenidos y qué tipo de operaciones se podrán realizar sobre ellos. Java es un lenguaje estáticamente tipado, por lo que todas las variables deberán ser declaradas con anterioridad a su uso.

Un tipo de dato *primitivo* es aquel que ya viene definido por defecto en el lenguaje y que, por lo tanto, debe ser referenciado mediante una palabra reservada.

Cuadro 4.2
Tipos de datos primitivos

Nombre	Tipo de dato	Ocupa	Rango	Valor defecto
`byte`	Entero (signo)	1 byte	-128 a 127	0
`short`	Entero (signo)	2 bytes	-32768 a 32767	0
`int`	Entero (signo)	4 bytes	-2^{31} a $2^{31}-1$	0
	Entero		0 a $2^{32}-1$	
`long`	Entero (signo)	8 bytes	-2^{63} a $2^{63}-1$	0L
	Entero		0 a $2^{64}-1$	
`float`	Decimal simple	4 bytes	Punto flotante 32-bit IEEE 754	0.0f
`double`	Decimal doble	8 bytes	Punto flotante 64-bit IEEE 754	0.0d
`char`	Carácter simple	2 bytes	'\u0000' a '\uffff' (65,535)	'\u0000'
`boolean`	Booleano	1 byte	true o false	false

4.8.1. Conversiones de tipo (*casting*)

Existe la posibilidad de transferir información de variables declaradas de un tipo a otro y viceversa, aunque no siempre es posible realizar este tipo de operación. Este paso es denominado *conversión*, *cast*, *casting*, *moldeado* o *tipado*.

Cuando se realice una conversión esta deberá evitar pérdida de información. Como norma general, las conversiones seguras serán aquellas que almacenen datos que ocupen un número determinado de *bytes* inferior a los del tipo de destino.

Cuadro 4.3
Conversiones seguras

Tipo de dato origen	Tipo de dato destino
`byte`	double, float, long, int, char, short
`char`	double, float, long, int
`short`	
`int`	double, float, long
`long`	double, float
`float`	double

En el caso de conversiones de tipos enteros a decimales es importante tener en cuenta que, aunque el sistema permite realizarlas y no anuncie ningún error, es posible encontrar fallos cuando se manejen valores muy altos.

También será posible realizar conversiones no seguras que, dependiendo de los valores que se vayan a manejar, pueden realizarse sin riesgo tal como puede verse en el siguiente ejemplo:

```
byte b = 'c';
```

b almacenará el código que representa a 'c'

A) Implícitas

Son realizadas de forma automática al realizar alguna operación de asignación y no declarar explícitamente la conversión. A continuación, puede verse un ejemplo de conversiones implícitas:

```
byte b = 1;
short s = b;
int i = s;
long l = i;
float f = l;
double d = f;
```

Algunas de las conversiones seguras mencionadas en el cuadro 4.3 no pueden realizarse de esta manera.

B) Explícitas

Se realizan anteponiendo al dato origen de la conversión el tipo de la variable destino. Este tipo de conversión permite hacer algunas de las conversiones seguras que no se puede hacer de forma implícita, así como las conversiones inseguras que tampoco pudiesen hacerse de la forma anteriormente citada (estas últimas pueden provocar pérdida de información y resultados erróneos). A continuación, puede verse un ejemplo de conversiones explícitas:

```
int i = 127;
byte b = (byte) i;

int i = 128;
byte b = (byte) i;
```

Tanto *i* como *b* almacenan 127

i almacena 128 y *b* -128

4.9. *String*

Puesto que normalmente son cadenas de caracteres con lo que se trabaja en los diferentes programas, almacenar y manejar caracteres aislados de tipo *char* no llega a ser del todo práctico en la mayor parte de los casos. En el lenguaje Java las cadenas de caracteres se trabajan con objetos de la clase *String*. Esta clase implementa gran cantidad de métodos que permiten trabajar con las cadenas de manera cómoda y sin necesidad de implementar nuestros propios métodos.

Existen varias formas de crear un *String*, aunque la más directa y sencilla es mediante una simple asignación de un texto encerrado entre comillas dobles. Los caracteres que se podrán utilizar serán exactamente los mismos que en el tipo primitivo *char*. En el capítulo 8 (Estructuras de almacenamiento) se abordará en detalle este tipo de dato. A continuación, se muestra un ejemplo simple de declaración de *Strings*:

```
String a = "esto es una prueba";
String b = a;
```

4.10. Clases envoltorio

Una clase envoltorio (*Wrapper Class*) es aquella cuyo objetivo es crear objetos que contengan un dato de tipo primitivo y proveer de métodos que faciliten su manejo. Algunos métodos están disponibles para cualquier clase; otros únicamente están disponibles en algunas de ellas.

Cuadro 4.4
Correspondencia tipo primitivo – Wrapper Class

Primitivo	boolean	byte	char	double	float	int	long	short
Wrapper	Boolean	Byte	Character	Double	Float	Integer	Long	Short

Cuadro 4.5
Métodos de cualquier Wrapper Class

`toString()`	Retorna un String cuyo literal es el valor del dato primitivo.
`x.compareTo(y)`	Retorna 0 si los datos contenidos en *x* e *y* son iguales. Retorna 1 si el valor de *x* es mayor al de *y*. Retorna -1 si el valor de *x* es menor al de *y*.
`x.equals(y)`	Retorna *true* si los datos contenidos en *x* e *y* son iguales y *false* en caso contrario.

Investiga

Busca información sobre los diferentes métodos que ofrecen las clases envoltorio y responde a las siguientes cuestiones:

¿Hay algún método que permita saber si un dato es o no un número?

¿Hay algún método que permita obtener un entero a partir de un decimal?

¿Hay algún método que retorne un objeto de clase envoltorio a partir del valor que represente un literal?

4.10.1. Autoboxing - Unboxing

Dado que existe correspondencia directa entre tipos primitivos y clases envoltorio, a partir del *JDK 5* se incluyó la conversión automática entre ambos tipos:

- Si la conversión es de tipo primitivo a envoltorio, se denomina *autoboxing*.
- Si la conversión es de tipo envoltorio a primitivo, se denomina *unboxing*.

A continuación, puede verse un ejemplo *autoboxing* y *unboxing*:

```
Integer a = 11, b = 20;  // autoboxing
int c = b - a;           // unboxing
```

4.11. Secuencias de escape

El lenguaje de programación Java permite el uso de secuencias de escape especiales que pueden emplearse tanto en *Strings* como en variables del tipo primitivo *char*. Este tipo de secuencias hace que se trate al carácter contenido en la secuencia como un literal sin significado alguno o justo todo lo contrario, permitiendo que sea reemplazado por otro contenido.

Cuadro 4.6
Secuencias especiales de escape

`\b`	Retrocede un espacio	`\r`	Retorno de carro
`\t`	Tabulación	`\"`	Comilla doble
`\n`	Nueva línea	`\'`	Comilla simple
`\f`	Salto de página	`\\`	Barra invertida

Para saber más

Como carácter válido para literales (*String* y *char*) u otra parte del código del programa (nombres de métodos, clases o campos) podrá emplearse cualquier carácter Unicode (UTF-16) en forma de secuencia de escape. La codificación de este carácter se realizará conforme al patrón \uDDDD, en donde DDDD es el código correspondiente al carácter. En la URL https://www.fileformat.info/info/charset/UTF-16/list.htm podrás acceder a un listado completo de caracteres UTF-16 con sus códigos correspondientes (los últimos 4 dígitos).

4.12. Comentarios

Los comentarios son textos que pueden embeberse en el código fuente para explicar su funcionalidad o para preparar la posterior generación de la documentación del código. Son ignorados por el compilador (no afecta al tamaño del archivo ejecutable).

En Java existen tres tipos:

- ✓ De *una línea*: comienzan con la secuencia //. Empleados para añadir explicaciones al código.
- ✓ De *varias líneas*: comienzan con la secuencia /* y finalizan con la secuencia */. Empleados para añadir explicaciones al código.
- ✓ De *documentación*: son un tipo especial de comentario de varias líneas. Comienzan con la secuencia /** y finalizan con la secuencia */. El programa *javadoc* utiliza estos comentarios para generar automáticamente la documentación del código.

Recurso digital

En el anexo web 4.1., disponible en www.sintesis.com y accesible con el código indicado en la primera página del libro, encontrarás información sobre cómo etiquetar un código para posteriormente generar la documentación asociada al mismo en el mismo formato que la documentación oficial de la API de Java.

4.13. Operadores y expresiones

Una vez se ha explicado cómo declarar e inicializar variables de tipos primitivos o que referencien a objetos de clases envoltorio, ha llegado el momento de realizar operaciones con ellas. Para ello se hace uso de los operadores, que son símbolos que permiten realizar operaciones sobre uno, dos o tres operandos. Tal como se explicó en el capítulo 1 (Introducción a la programación), existen varias categorías de operadores.

Cuadro 4.7
Resumen operadores

Binarios					
Aritméticos		**Relacionales**		**Lógicos**	
+	Suma en números y concatenación en Strings	==	Igual	&&	y lógico (conjunción)
-	Resta	<	Menor	||	o lógico (disyunción)
*	Multiplicación	≤	Menor o igual	**Especiales**	
/	División real	>	Mayor	=	Asignación
%	Resto o módulo	≥	Mayor o igual	instanceof	Objeto es de tipo
		!=	Distinto		
Unarios					
+	Indica valor positivo	++	Incrementa en 1	!	Negación lógica
-	Niega la expresión	--	Decrementa en 1		

[.../...]

Cuadro 4.7 (cont.)

Bit					
A<<B	Desplaza A a la izquierda B posiciones	A&B	Operación AND a nivel de bits	A^B	Operación XOR a nivel de bits
A>>B	Desplaza A a la derecha B posiciones (tiene en cuenta el signo)	A|B	Operación OR a nivel de bits	~A	Complemento de A a nivel de bits
A>>>B	Desplaza A a la derecha B posiciones (no tiene en cuenta el signo)				

4.13.1. Precedencia de operadores

Los operadores tienen un orden de precedencia, de manera que los operadores con más alto nivel de precedencia son evaluados antes que aquellos que tienen un nivel de precedencia menor. Aquellos operadores que tienen el mismo nivel de precedencia y aparecen a la vez en la misma expresión son evaluados según el orden en el que aparecen (de izquierda a derecha). Todos los operadores binarios se evaluarán de izquierda a derecha excepto el de asignación, que se evaluará de derecha a izquierda. En el cuadro 4.8 se resume la precedencia de operadores en la que el orden de aparición se corresponde directamente con una mayor precedencia.

Cuadro 4.8
Precedencia de operadores

Precedencia
expr++ expr--
++expr --expr +expr -expr ~ !
* / %
+ -
<< >> >>>
< > <= >= instanceof
== !=
&
^
|
&&
||
= += -= *= /= %= &= ^= |= <<= >>= >>>=

Resumen

- Los programas informáticos se desarrollan en soluciones que contienen proyectos en los que se almacenan los archivos de código.
- Antes de comenzar a programar es importante conocer las reglas de construcción de identificadores, así como las palabras reservadas que no se podrán utilizar.
- El código se organiza por paquetes, los cuales contendrán clases.
- Paquetes, clases, variables, constantes y métodos deben ser nombrados con identificadores válidos.
- Existen varias convenciones que dan ideas de cómo llevar a cabo un nombrado lógico.
- En Java existen 8 datos primitivos y 8 clases envoltorio que facilitan su uso, además del tipo *String* que se emplea para manejar literales.
- Las secuencias de escape son de gran utilidad a la hora de manejar literales.
- Se puede utilizar una gran variedad de operadores y cada uno será evaluado en un orden.

Ejercicios propuestos

1. Enumera y describe los diferentes tipos de comentarios que se pueden encontrar en un código fuente Java.
2. Describe la diferencia entre la conversión implícita y explícita. Propón ejemplos.
3. ¿Qué diferencia hay entre *solución* y *grupo de proyectos*?
4. Enumera y describe los diferentes tipos de datos primitivos y elabora un resumen de las conversiones que se pueden realizar sin pérdida de información.
5. ¿En qué consiste la notación húngara? Pon un ejemplo.
6. A partir del siguiente listado de palabras, identifica aquellas que sean identificadores válidos: $hora, 1LoveYou, Calcula_fecha, calcula Fecha, calculaFecha, IVA, iva%.

Supuestos prácticos

1. Escribe el código de una nueva clase en un nuevo proyecto sobre tu IDE. Establece un identificador válido para el proyecto (*lowercase*) y para la clase (*lowerCamelCase*).
2. Haz que la clase sea la clase principal del proyecto implementando el método main.
3. Define en el método principal dos constantes, una para almacenar un número entero (primitivo) y otra para almacenar literales de más de 1 carácter. Inicialízalas a 1008 y "Saludos" respectivamente.
4. Declara en el método principal dos variables de tipo envoltorio para almacenar números enteros. Inicializa la primera de ellas al valor de la constante y una vez lo hayas hecho, inicializa la siguiente al contenido de la primera variable (independientemente del valor que tenga).
5. Haciendo uso de la llamada *System.out.print(s);* imprime por pantalla el contenido de las variables y el literal.

Actividades de autoevaluación

1. Señala la afirmación falsa:
   - ☐ *a)* Un proyecto es un conjunto de carpetas y archivos organizados conforme a cierto criterio que permita llevar un orden lógico.
   - ☐ *b)* Un proyecto solo puede contener archivos de código.
   - ☐ *c)* Una solución es un conjunto de uno o más proyectos relacionados entre sí y se emplea comúnmente en la tecnología *.NET*.
   - ☐ *d)* En *NetBeans* se utilizan los *Project Groups*.
2. ¿Qué valor tomará la variable b tras la ejecución → *int b = 9 <<2;*?
   - ☐ *a)* 7.
   - ☐ *b)* 11.
   - ☐ *c)* 34.
   - ☐ *d)* 36.
3. ¿Qué valor tomará la variable b tras la ejecución → *int b = 16 & 8;*?
   - ☐ *a)* 8.
   - ☐ *b)* 16.
   - ☐ *c)* 24.
   - ☐ *d)* Todas las opciones anteriores son incorrectas.

4. ¿Cuál de las siguientes es una afirmación falsa acerca de un paquete?
   - ☐ *a)* Permite agrupar clases con características comunes.
   - ☐ *b)* En su declaración su nombre puede contener espacios.
   - ☐ *c)* Proporciona un mecanismo de control de acceso.
   - ☐ *d)* Para acceder a clases que estén en paquetes diferentes al actual se puede importar el paquete.

5. ¿Qué resultado contendría la variable tras la siguiente instrucción → *a = b++ + 13;*?
   - ☐ *a)* El valor de (b) + 13.
   - ☐ *b)* El valor de (b+1) + 13.
   - ☐ *c)* El valor de (b+2) + 13.
   - ☐ *d)* Todas las opciones anteriores son incorrectas, puesto que se produce un error de compilación.

6. ¿Cuál de las siguientes conversiones no es segura?
   - ☐ *a)* *short a int.*
   - ☐ *b)* *int a long.*
   - ☐ *c)* *double a byte.*
   - ☐ *d)* *float a double.*

7. El modificador empleado para compilar un programa y depositar los ficheros .class en una carpeta es:
   - ☐ *a)* –f.
   - ☐ *b)* –p.
   - ☐ *c)* –d.
   - ☐ *d)* No existe dicha opción.

8. ¿Cuál de las siguientes secuencias de escape permite insertar un retorno de carro?
   - ☐ *a)* \n
   - ☐ *b)* \b
   - ☐ *c)* \r
   - ☐ *d)* \t

9. ¿Cuál de las siguientes no es una regla válida para construir un identificador?
   - ☐ *a)* Debe contener, al menos, un carácter.
   - ☐ *b)* No debe superar los 8 caracteres.
   - ☐ *c)* El primer carácter no podrá ser un dígito.
   - ☐ *d)* Los caracteres válidos serán letras (a-Z), dígitos (0-9), guion bajo (_) o dólar ($).

10. Identifique el operador unario entre los siguientes:
    - ☐ *a)* *
    - ☐ *b)* !
    - ☐ *c)* /
    - ☐ *d)* %

SOLUCIONES:

1.	a	**b**	c	d	5.	**a**	b	c	d						
2.	a	b	c	**d**	6.	a	b	**c**	d	9.	a	**b**	c	d	
3.	a	b	c	**d**	7.	a	b	**c**	d	10.	a	**b**	c	d	
4.	a	**b**	c	d	8.	a	b	**c**	d						

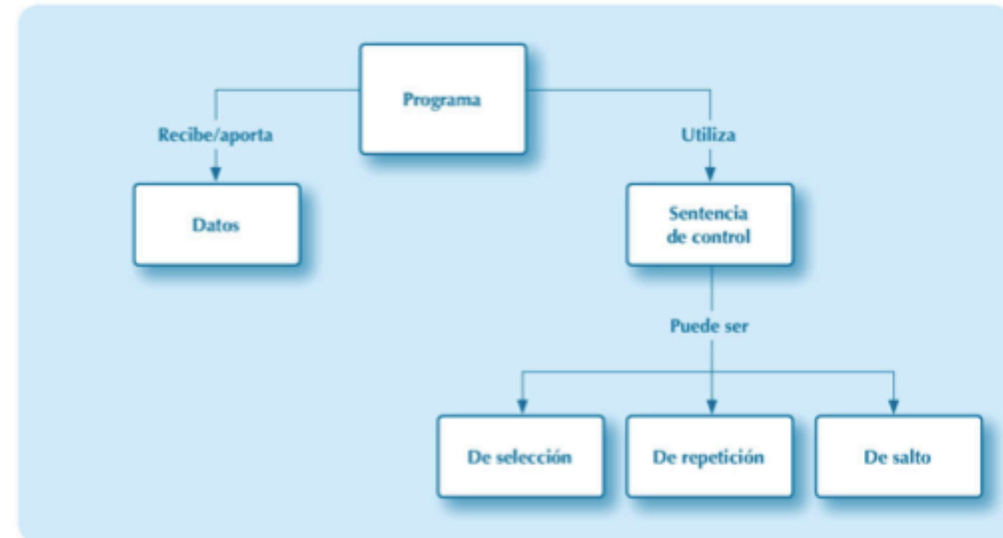
5

Estructuras de control

Objetivos

- ✓ Conocer los mecanismos básicos de entrada y salida de datos desde consola en un programa.
- ✓ Comprender el manejo de las diferentes estructuras de selección, repetición y salto.
- ✓ Desarrollar un programa que evalúe datos de entrada y devuelva el resultado correspondiente.

Mapa conceptual



Glosario

Argumento. Valor variable pasado a una función o método (sirva de ejemplo el método main) para que realice una tarea en función de este.

Array. Almacenamiento en memoria de un conjunto de datos ordenado del mismo tipo. También se conoce como vector o matriz de una dimensión.

Bug. Término genérico que define aquel comportamiento que tiene un programa y para el que no fue diseñado.

Iteración. En programación se denomina *iteración* a cada repetición de una sentencia o conjunto de ellas.

Parámetro. Variable declarada en la misma declaración de una función o método (sirva de ejemplo el método *main*), que recoge el valor de los argumentos con los que se realizó la invocación.

Password. Contraseña, código de acceso o clave de paso que es necesario conocer para acceder a una determinada información o sistema.

Sentencia condicional. Sentencia que transfiere el control del programa a otra línea de ejecución en función de una o varias condiciones.

Sentencia incondicional. Sentencia que transfiere el control del programa a otra línea de ejecución sin tener en cuenta ninguna condición.

Sentencia. Instrucción perteneciente a un lenguaje de programación que es ejecutada en un programa informático.

Stream. Flujo de datos empleado para la salida y entrada de información.

String. Secuencia de caracteres Unicode que, a diferencia de otros lenguajes, en Java es representada mediante una clase de mismo nombre que facilita su manejo.

Token. Cadena de caracteres que tiene un significado especial y que viene contenida en una cadena de mayor tamaño.

5.1. Introducción

Los programas son de muy diverso tipo y, debido a ello, pueden tomar los datos con los que trabajan desde varios orígenes; además, también son diseñados para mostrar los resultados de distintas maneras. Pero independientemente de cómo obtengan los datos con los que trabajan, la ejecución de las instrucciones contenidas en ellos es secuencial y sigue un orden que va desde las primeras líneas al final del programa. Gracias a las sentencias de control será posible alterar el orden de ejecución según la lógica que sea necesario aplicar.

En este capítulo se explicará la forma básica de comunicación con un programa Java, así como las sentencias de control que permiten trabajar con la información recibida.

5.2. Entrada y salida de información

Existen varias alternativas para comunicarse con los programas Java. En los apartados siguientes se explicarán la entrada de información por argumentos (en la llamada) y la entrada-salida de información por el terminal.

5.2.1. Entrada por argumentos

Todo programa Java tiene como punto de partida el método *main* declarado en la clase principal. A partir de ese método el programa realizará las tareas correspondientes hasta que finalice el trabajo o se detenga, bien por causas controladas o ajenas a la normal ejecución de este.

El programa *main* se declara como un método estático (no necesita de una instancia de la clase que lo contiene para poder ser invocado), público (es accesible desde cualquier lugar) y con el parámetro *args* que contendrá los argumentos pasados cuando el programa se invoque con argumentos. Dicho parámetro es del tipo *array*, un tipo complejo no visto hasta ahora que se corresponde con una colección de elementos y que será explicado en capítulos posteriores.

Hasta ahora únicamente se han ejecutado programas que establecían los datos con los que trabajan desde el mismo programa y sin ningún paso de argumentos o interacción con el usuario.

Para saber más

Los términos *parámetro* y *argumento* a menudo se utilizan indistintamente, pero en realidad no son sinónimos. Un argumento es el dato que se pasa cuando se hace la llamada a un método, mientras que un parámetro es la variable en la que se almacena el dato recibido desde la invocación.

Para pasar argumentos a un programa es necesario añadirlos a la llamada (tanto si se ejecutó de forma simple como si venía contenido en un *jar* ejecutable). Podrán enviarse tantos argumentos como sea necesario, cada uno de ellos separado por un espacio. Dentro del método *main* podrán recuperarse como *Strings* con *args[0]* para el primer argumento, *args[1]* para el segundo… y así sucesivamente.

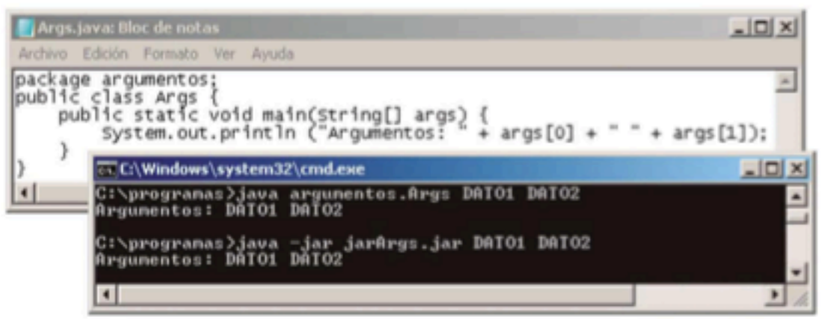


Figura 5.1
Paso de información por argumentos desde línea de comandos.

Si se desea trabajar desde *NetBeans* con argumentos, hay que acceder a las *propiedades del proyecto*, categoría *Run* y en la sección *Arguments* especificar los argumentos de la misma forma que se haría por el terminal.

Actividad propuesta 5.1

¿Es posible enviar la frase "Hola mundo" con un único argumento a un programa Java desde la terminal? ¿Y desde *NetBeans*? Justifica tu respuesta.

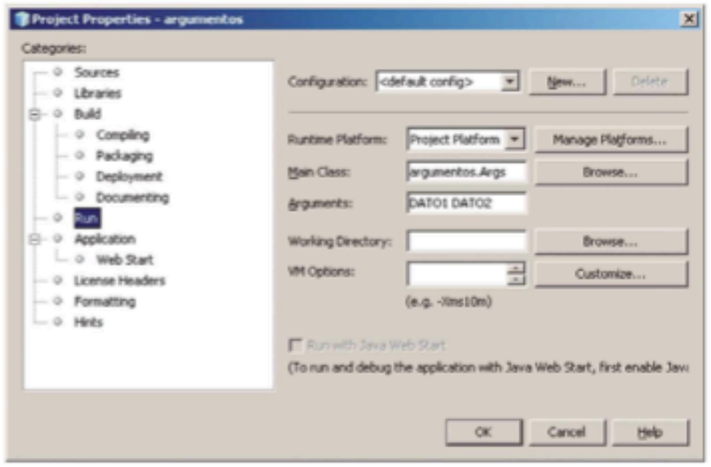


Figura 5.2
Paso de información por argumentos desde *NetBeans*.

5.2.2. Entrada y salida por consola

Mediante la invocación con argumentos se puede pasar información al programa que podrá recoger para comenzar la ejecución. Sin embargo, solo con esto no será posible implementar programas interactivos. En Java la entrada y salida de datos se realiza mediante flujos de datos (*Streams*). Este mecanismo, que ha estado presente desde la primera versión de Java, proporciona una serie de flujos de datos estándar que estarán disponibles mientras el programa esté en ejecución y listos para proporcionar datos. Son los siguientes:

1. *Flujo estándar de salida System.out*: se trata de un objeto de la clase *PrintStream* que por defecto es la pantalla. Su manejo no presenta ninguna complicación, dado que ya ha sido utilizada en ejemplos anteriores. Para mostrar datos por la consola se puede hacer uso de los métodos:

   a) *System.out.write (datos)* → Imprime los *bytes* enviados como argumento por el flujo estándar.
   b) *System.out.print (obj)* → La información que se le envía como argumento se transforma a un conjunto de *bytes* y posteriormente se imprime haciendo uso del método *System.out.write*.
   c) *System.out.println (obj)* → Igual que el anterior añadiendo un salto de línea al final.
   d) *System.out.printf (String cadena, [tipo argumento1] … [,tipo argumentoN])* → Imprime *cadena* incluyendo los *argumentos* en las especificaciones de formato. En el caso de que el número de argumentos sea superior al de especificaciones de formato en cadena, los argumentos sobrantes se ignoran.

2. *Flujo estándar de salida de error System.err*: similar a *System.out* pero para salida de error.

3. *Flujo estándar de entrada System.in*: se trata de un objeto de la clase *InputStream* que por defecto es el teclado. Para leer datos desde el teclado se puede hacer uso de los métodos:

   a) *System.in.read (datos)* → Almacena en *datos* los caracteres leídos desde el teclado. Datos debe ser declarado previamente como un *array* de *bytes*.
   b) *System.in.read ()* → Retorna el *byte* siguiente de datos desde el flujo de entrada como un entero comprendido entre 0 a 255 (se debe convertir a *char* mediante un *casting*).

A continuación, puede verse un ejemplo de Entrada/Salida mediante flujos estándar:

```
int c1, c2, c3;
System.out.println("Introduzca 3 caracteres:");
c1 = System.in.read();
c2 = System.in.read();
c3 = System.in.read();
System.out.write(c1);
System.out.print((char)c2);
System.out.println((char)c3);
```

Otro mecanismo mucho más cómodo para realizar la lectura de información desde un *Stream* es la clase *Scanner*. Esta clase, que fue incluida a partir de *Java 1.5*, es capaz de analizar tipos primitivos y dividir la entrada en *tokens* que, por defecto, vienen separados por espacios. Algunos de los métodos más conocidos son:

- ✓ *nextLine ()* → Retorna la línea completa como un *String*.
- ✓ *next ()* → Retorna el siguiente *token* como un *String*.
- ✓ *nextXXXX ()* → Retorna el siguiente *token* como si fuese de tipo XXXX (siendo *XXXX Byte, Int, Double…*).
- ✓ *hasNext ()* → Devuelve *true* si el escáner puede devolver otro *token*.
- ✓ *hasNextInt ()* → Devuelve *true* si el próximo *token* puede interpretarse como un valor de tipo *XXXX* utilizando el método *nextXXXX()*.

A continuación, puede verse un ejemplo de entrada/salida utilizando la clase *Scanner*:

```
package entradasalida ;
import java.util.Scanner;
public class EntradaSalida {
   public static void main(String[] args) {
        System.out.println("Introduzca el texto a reproducir:");
        Scanner entradaEscaner = new Scanner (System.in);
        String entradaTeclado = entradaEscaner.nextLine ();
        System.out.println (entradaTeclado);
 }
}
```

También puede ser de gran utilidad la clase *Console* que incluye funcionalidad para lectura de *passwords* ocultando los caracteres tecleados. Está disponible desde *Java 1.6* y sus métodos principales son:

- *readLine ()* → Retorna la línea completa como un *String*.
- *readPassword ()* → Devuelve un *array* de caracteres pero ocultando los caracteres que son introducidos por teclado.
- *printf (String cadena, [tipo argumento1] … [,tipo argumentoN])* → Imprime texto por consola de forma similar a *System.out.printf*.

En la figura 5.3 puede verse el resultado de ejecutar el código del siguiente ejemplo en el que se utiliza la clase *Console* para la entrada y salida de datos.

```
package entradasalida;
import java.io.Console;
public class EntradaSalida {
    public static void main(String[] args) {
        Console c = System.console();
        if (c == null)
                System.err.println("No hay consola.");
        else {
            System.out.println("Introduzca su nombre: ");
            String nombre = c.readLine();
            System.out.println("Hola "+nombre);
            System.out.println("Introduzca su password: ");
            String password = String.valueOf(c.readPassword());
            System.out.println("Su Password es: "+password);
        }
    }
}
```

```
C:\Windows\system32\cmd.exe
C:\Users\[redacted]\Documents\NetBeansProjects\entradasalida\build\classes>java entr
adasalida.EntradaSalida
Introduzca su nombre:
Pedro
Hola Pedro
Introduzca su password:

Su Password es: Síntesis↑

C:\Users\genete\Documents\NetBeansProjects\entradasalida\build\classes>_
```

Figura 5.3
Ejemplo de Entrada/Salida utilizando la clase *Console*.

Recurso web

En la URL https://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html#syntax puedes encontrar más información sobre los especificadores de formato para cadenas Java.

Actividades propuestas

5.2. Busca información sobre el método *System.out.flush* e indica un caso en el que pueda ser de utilidad.

5.3. A la clase *Scanner* se le puede indicar, mediante cadenas o expresiones regulares, cómo separar los *tokens*. Investiga sobre cómo se forman las expresiones regulares y construye algunas para reconocer los patrones de un DNI, número de teléfono y dirección de correo electrónico.

5.3. Estructuras de selección (alternativa)

Todo lenguaje de programación ofrece estructuras de control y selección con las que escribir el código. A continuación, se van a exponer las sentencias de control que son la materialización en el lenguaje Java de las estructuras de control expuestas en el capítulo 1 (Introducción a la programación).

5.3.1. *if, else, else if*

La sentencia condicional *if* es la sentencia de control más básica de cualquier lenguaje de programación. Permite evaluar una condición y actuar en consecuencia en función de su resultado. Existen algunas variantes de esta sentencia condicional:

1. *if* → Se evalúa la condición encerrada entre paréntesis y si el valor de retorno es *true*, se ejecutará el bloque de instrucciones encerrado entre llaves. En caso contrario se saltará a la siguiente instrucción.
2. *if-else* → Similar a la sentencia *if* anterior, pero en este caso también es capaz de ejecutar un bloque de instrucciones si la condición evaluada retorna *false*.
   Además de las sentencias anteriores hay que destacar la existencia del operador ternario que permite realizar un *if-else* simple en una única línea de código.
3. *if-else if* → La sentencia *if-else* ejecuta un segundo bloque de instrucciones en el caso de que la condición inicial devolviera *false* y esto lo hacía sin evaluar nuevas condiciones. Si se necesita seguir evaluando condiciones una vez que la primera no se ha cumplido, es necesario hacer el *else* condicional. Para ello se irán encadenando diferentes bloques *if-else*.

Cada bloque de instrucciones puede contener a su vez nuevas sentencias condicionales. De esta manera se pueden ir anidando unos a otros. A continuación pueden verse varios ejemplos de sentencias condicionales *if*:

	EQUIVALENTES ↓	

`if (a>b) {`
`  mayor = a;`
`  menor = b;`
`}`
	`if (x>y)`
	`  mayor = x;`
	`else`
	`  mayor = y;`
		`if (dia=="lunes") {`
		`  …`
		`} else if (dia=="martes") {`
		`  …`
		`} else if (dia=="domingo") {`
		`  …`
		`} else {`
		`  …`
		`}`

	`mayor = (x>y)?x:y;`	

5.3.2. *switch*

Muchas de las soluciones que se implementan con la sentencia *if-else if* pueden realizarse de forma más clara con *switch*. La sentencia condicional *switch* permite evaluar una variable y ejecutar un bloque de sentencias en función del valor que tome. Puede incluirse, es opcional, el valor *default* que será el que determinará las instrucciones a ejecutar si no hubo coincidencia con ninguno de los casos anteriores. Esta sentencia de control es capaz de trabajar con:

- ✓ Tipos primitivos *byte*, *short*, *char*, *int*.
- ✓ Tipos *enumerados*.
- ✓ *Strings*.
- ✓ Clases envoltorio *Character*, *Byte*, *Short* e *Integer*.

A continuación, puede verse un ejemplo de uso de sentencias condicionales *switch* en el que se muestran los días que faltan hasta el fin de semana:

```
int dia = 1; // 1-L, 2-M, 3-X, 4-J, 5-V
switch (dia) {
   case 1:  System.out.println("Lunes");
   case 2:  System.out.println("Martes");
   case 3:  System.out.println("Miércoles");
   case 4:  System.out.println("Jueves");
   case 5:  System.out.println("Viernes");
            break;
   default: System.out.println("Fin de semana");
            break;
}
```

Como se habrá podido comprobar, en los bloques de instrucciones de cada opción es posible encontrar un *break*. Esta instrucción provoca una interrupción en el flujo de ejecución del bloque en el que se encuentre y fuerza a que se ejecute la instrucción siguiente al bloque interrumpido. Si no se introduce esta instrucción, se seguirán ejecutando las instrucciones que encuentre a continuación. La instrucción *break* se verá en detalle en apartados posteriores.

5.4. Estructuras de repetición (iterativa)

Las estructuras de repetición, o bucles como suelen denominarse en jerga informática, permiten ejecutar bloques de instrucciones durante un número determinado o indeterminado de iteraciones.

5.4.1. *for*

La sentencia de control *for* proporciona una forma de iterar sobre un determinado rango de valores. Está formada por una *expresión de inicialización* (que únicamente se ejecutará al comienzo), una *expresión de finalización* que causará la finalización del bucle cuando retorne *false* y una *expresión de incremento/decremento* que es invocada cada *iteración* del bucle. Si se omiten las 3 expresiones anteriores, será un bucle infinito. A continuación, se muestran varios ejemplos de sentencias *for*.

```
for (int i=1; i<4; i++) {
    System.out.println("Incrementando: " + i);
}
for (int j=3; j>0; j--) {
    System.out.println("Decrementando: " + j);
}
for ( ; ; ) {
    System.out.println("Bucle infinito");
}
```

```
Incrementando 1
Incrementando 2
Incrementando 3
Decrementando 3
Decrementando 2
Decrementando 1
Bucle infinito
Bucle infinito
...
```

La declaración de la variable empleada en la *expresión de incremento/decremento* podrá realizarse fuera de la sentencia iterativa o bien estar contenida de forma implícita en la *expresión de inicialización* (tal como se ha hecho en los ejemplos). La diferencia está en que, si se declara fuera la variable, esta seguirá existiendo hasta que finalice el bloque en el cual ha sido declarado el *for*, mientras que, si se hace en el propio *for*, se destruirá una vez finalice el bucle:

```
public static void main(String[] args) {
    int i=0;
    for (i=0; i<4; i++) {}
    System.out.println("Valor de i: " + i);
}
```

```
run:
Valor de i: 4
```

5.4.2. *while*

La sentencia *while* evalúa la condición y si retorna *true* ejecuta el bloque de instrucciones. Las sentencias contenidas puede que no se ejecuten nunca si la condición inicialmente retorna un

valor *false*. Salvo que se quiera implementar un bucle infinito, las instrucciones contenidas en el bloque deberán alterar el resultado de la condición en algún momento. En el ejemplo siguiente puede verse cómo utilizar una sentencia *while* y el resultado producido:

```
int i = 1;
while (i < 4) {
    System.out.println("Iteración: " + i);
    i++;
}
```

```
Iteración: 1
Iteración: 2
Iteración: 3
```

5.4.3. *do while*

Esta sentencia es parecida a *while* pero con la diferencia de que la sentencia *do while* primero ejecuta el bloque de instrucciones y posteriormente evalúa la condición. Por tanto siempre ejecutará el bloque de instrucciones como mínimo una vez. A continuación se muestra un ejemplo de uso de la sentencia *do while* y del resultado que produciría:

```
int i = 1;
do {
    System.out.println("Iteración: " + i);
    i++;
} while (i < 1);
```

```
Iteración: 1
```

5.5. Estructuras de salto incondicional

Este tipo de estructuras de control permiten modificar el flujo normal de ejecución del programa para realizar saltos a diferentes partes del código fuente. Aunque son expuestas en este apartado, su uso está totalmente desaconsejado desde el punto de vista de la programación estructurada, puesto que complica el seguimiento y trazabilidad de los programas. Este tipo de estructuras son perfectamente reemplazables por las expuestas anteriormente (*secuencial*, *alternativa* e *iterativa*).

5.5.1. *break continue*

La sentencia *break* permitirá detener la ejecución de un bucle y saltar a la siguiente instrucción del programa tras el bucle. En los ejemplos siguientes puede verse cómo utilizar la sentencia de salto *break* dentro de una sentencia *while*:


```
while (true) {
    System.out.println("Dentro bucle 1");
    while (true) {
        System.out.println("Dentro bucle 2");
        break;
    }
}
```

```
Dentro bucle 1
Dentro bucle 2
Dentro bucle 1
Dentro bucle 2
.
.
.
```

```
int j=1;
while (true){
    System.out.println("Iteración " + j);
    if (j==3) break;
    j++;
}
System.out.println("Tras 3 iteraciones salgo");
```

```
Iteración 1
Iteración 2
Iteración 3
Tras 3 iteraciones salgo
```

La sentencia *continue* permite detener únicamente la *iteración* actual y pasar a la siguiente *iteración* del bucle sin salir de él (salvo que el propio bucle haya llegado a su fin). En los ejemplos mostrados a continuación podrá verse cómo utilizar la sentencia de salto *continue*:

```
while (true) {
    System.out.println("Dentro bucle 1");
    while (true) {
       System.out.println("Dentro bucle 2");
       continue;
    }
}
```

```
Dentro bucle 1
Dentro bucle 2
Dentro bucle 2
.
.
.
```

```
for(int j=1; j<=3; j++){
    System.out.println("Iteración " + j);
    if (j<3) continue;
    System.out.println("Saliendo del bucle");
 }
System.out.println("Fuera del bucle");
```

```
Iteración 1
Iteración 2
Iteración 3
Saliendo del bucle
Fuera del bucle
```

Normalmente, las instrucciones *break* y *continue* van dentro de una sentencia *if* que indicará cuándo detener el bucle en función de una condición. La sentencia *break* puede encontrarse dentro de un bucle o en una sentencia *switch*, mientras que la sentencia *continue* debe ir siempre dentro de un bucle.

5.5.2. *break continue* con etiquetas

Tanto *break* como *continue* pueden usar etiquetas para que, en lugar de salir del bucle o saltar a la siguiente *iteración* como ocurre cuando se usan sin ninguna etiqueta, continúen por la sección

del código identificada por la etiqueta. Las etiquetas no pueden colocarse en cualquier parte del código, por lo que únicamente se podrán hacer saltos a bucles dentro de los que nos encontremos. Las etiquetas se declaran como *nombreEtiqueta:*. A continuación, se muestra un ejemplo de uso de estas junto con sentencias de salto *break-continue*:

```
A: while (true) {
    System.out.println("Dentro de A");
    B: while (true) {
        System.out.println("Dentro de B");
        break A;
    }
}
C: while (true) {
    System.out.println("Dentro de C");
    D: while (true) {
        System.out.println("Dentro de D");
        continue C;
    }
}
```

```
Dentro de A
Dentro de B
Dentro de C
Dentro de D
Dentro de C
Dentro de D
Dentro de C
Dentro de D
Dentro de C
Dentro de D
Dentro de C
.
.
.
```

Actividad propuesta 5.4

Entre las palabras reservadas del lenguaje Java que fueron expuestas en el apartado 4.3 se encuentra *goto*. Busca información sobre ella y propón, si es viable, un ejemplo de uso.

5.6. Prueba y depuración de programas

Todo programa deberá ser testado mediante un conjunto de pruebas que garanticen que este sea operativo, cumpla con los requisitos para los que fue desarrollado e incluya comentarios lo suficientemente aclaratorios como para facilitar su comprensión a otros desarrolladores. Ligado a este proceso de pruebas está el término *bug*, el cual define al comportamiento anómalo de un programa (para el que este no fue diseñado).

Al proceso mediante el cual se corrigen estos errores del programa se le conoce como depuración e implica una serie de tareas:

- La inspección del código del programa o algoritmo mediante el seguimiento del flujo del mismo, intentando predecir los resultados cuando esto sea posible.
- La utilización de baterías de prueba que sometan al programa a un trabajo exigente, de manera que ante una ejecución exitosa se pueda pronosticar un correcto funcionamiento en un escenario real.
- La prueba de todos y cada uno de los módulos de los que pueda constar el programa (pruebas unitarias).
- La corrección de todos y cada uno de los errores detectados, teniendo en cuenta que ante cada modificación del código pueden surgir nuevos errores.

Investiga

JUnit y *TestNG* son dos herramientas que se pueden emplear para llevar a cabo pruebas unitarias de los diferentes módulos de un programa. Investiga cómo hacer uso de ellas sobre el IDE NetBeans.

Resumen

- Un programa puede recibir datos de entrada a través de sus parámetros o bien a través de flujos de entrada.
- Por defecto, todo programa dispone durante su ejecución de 2 flujos de salida (normal y errores) y uno de entrada.
- Existen diferentes sentencias de control (*if* y *switch*), de repetición (*for*, *while* y *do while*) y de salto incondicional (*break* y *continue*), estas últimas desaconsejadas.

Ejercicios propuestos

1. Describe los métodos empleados para imprimir y leer información desde consola.
2. ¿Es posible separar la salida estándar de la salida de error? Razona tu respuesta.
3. ¿Por qué motivo no se recomienda el uso de estructuras de salto incondicional?
4. Compara las instrucciones *break* y *continue*. Propón un ejemplo.
5. Realiza un estudio sobre los diferentes métodos *next* de la clase *Scanner* y propón ejemplos de uso.

Supuestos prácticos

1. Implementa un programa que pida al usuario por teclado sus datos personales en orden *apellido1-apellido2-nombre* y los muestre por pantalla en orden *nombre-apellido1-apellido2*.
2. Desarrolla un programa que, una vez ejecutado, lea información desde teclado hasta que reciba la cadena de entrada "FIN".
3. Implementa un programa que reciba como parámetro un dígito correspondiente a un año y calcula si es o no bisiesto.
4. Escribe un programa calculadora que reciba como parámetros dos enteros (*num1* y *num2*) desde su llamada y que los muestre por pantalla. A continuación, se deberá solicitar al usuario

que seleccione la operación matemática a realizar entre los dos números. Las operaciones que implementará serán suma, resta, multiplicación, división y potencia.

5. Desarrolla un programa que pregunte por pantalla un número N y que, una vez leído, imprima una pirámide N filas.

6. Utilizando la sentencia de control *switch*, implementa un programa que pida un número por teclado y que indique por pantalla a qué mes se corresponde.

7. Analiza el siguiente fragmento de código y describe su comportamiento:

```
System.out.println("Introduzca un texto: ");
while (true){
    int letra = System.in.read();
    System.out.print((char)letra);
}
```

Actividades de autoevaluación

1. ¿Cuál de las siguientes afirmaciones es falsa?
   - ☐ *a)* A un *jar* ejecutable se le envían los argumentos separados por espacios.
   - ☐ *b)* A un programa Java se le envían los argumentos separados por espacios.
   - ☐ *c)* Los parámetros que recibe un programa pueden ser de cualquier tipo primitivo.
   - ☐ *d)* El primer parámetro recibido se corresponde con *args[0]*.

2. ¿Qué instrucción imprime un salto de línea retorno de carro al final del texto que recibe?
   - ☐ *a)* *System.out.write.*
   - ☐ *b)* *System.out.print.*
   - ☐ *c)* *System.out.println.*
   - ☐ *d)* Todas las opciones anteriores son incorrectas.

3. ¿Cuál de los siguientes métodos admite especificaciones de formato?
   - ☐ *a)* *System.out.write.*
   - ☐ *b)* *System.out.printf.*
   - ☐ *c)* *System.out.print.*
   - ☐ *d)* *System.out.println.*

4. ¿Cuál de las siguientes no representa un objeto de la clase *PrintStream*?
   - ☐ *a)* *System.in.*
   - ☐ *b)* *System.out.*
   - ☐ *c)* *System.err.*
   - ☐ *d)* Niguna de las anteriores opciones representa objetos de la clase *PrintStream*.

5. ¿Qué clase de las siguientes facilita la lectura de contraseñas ocultando los caracteres introducidos?
   - ☐ *a)* *java.util.Scanner.*
   - ☐ *b)* *java.io.Console.*
   - ☐ *c)* *java.io.Scanner.*
   - ☐ *d)* *java.util.ConsoleSystem.out.*

6. ¿Con cuál de las siguientes sentencias se puede realizar un bucle infinito?
   - ☐ *a)* *while.*
   - ☐ *b)* *for.*
   - ☐ *c)* Las opciones *a)* y *b)* son correctas.
   - ☐ *d)* Las opciones *a)* y *b)* son incorrectas.

7. ¿Es posible hacer un bucle infinito con una sentencia do *while*?
   - ☐ *a)* Sí, haciendo que la condición evaluada de *while* siempre sea *true*.
   - ☐ *b)* Sí, haciendo que el bloque *do* que se ejecuta como mínimo una vez sea en sí mismo un bucle infinito.
   - ☐ *c)* Las opciones *a)* y *b)* son correctas.
   - ☐ *d)* Las opciones *a)* y *b)* son incorrectas.

8. Si se desea ejecutar un bloque de instrucciones hasta que se cumple una condición, pero que se ejecute como mínimo una vez, se deberá emplear:
   - ☐ *a)* *while.*
   - ☐ *b)* *do while.*
   - ☐ *c)* *for.*
   - ☐ *d)* Todas las opciones anteriores son correctas.

9. ¿Cuál de las siguientes es una afirmación falsa?
   - ☐ *a)* La sentencia *switch* puede llevar sección *default*.
   - ☐ *b)* La sentencia *switch* puede llevar *break* en sus secciones.
   - ☐ *c)* La sentencia *switch* siempre lleva sección *default*.
   - ☐ *d)* En ocasiones una sentencia *switch* puede reemplazar una sentencia *if-else if*.

10. ¿Cuáles son los tipos de datos capaces de manejar una sentencia *switch*?
    - ☐ *a)* Tipos *enumerados*, *Strings* y clases envoltorio *Character*, *Byte*, *Short* e *Integer*.
    - ☐ *b)* Tipos primitivos *byte*, *short*, *char*, *int*, *Strings* y clases envoltorio *Character*, *Byte*, *Short* e *Integer*.
    - ☐ *c)* Tipos primitivos *byte*, *short*, *char*, *int*, tipos enumerados, *Strings*.
    - ☐ *d)* Tipos primitivos *byte*, *short*, *char*, *int*, tipos enumerados, *Strings* y clases envoltorio *Character*, *Byte*, *Short* e *Integer*.

SOLUCIONES:

1.	a	b	**c**	d	5.	a	**b**	c	d						
2.	a	b	**c**	d	6.	a	b	**c**	d	9.	a	b	**c**	d	
3.	a	**b**	c	d	7.	a	b	**c**	d	10.	a	b	c	**d**	
4.	**a**	b	c	d	8.	a	**b**	c	d						

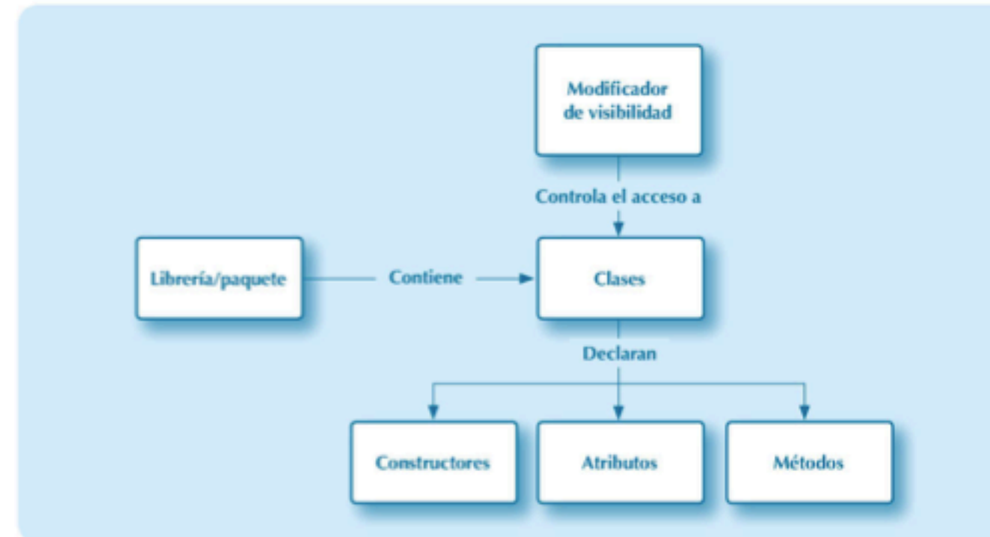
6

Clases y utilización de objetos

Objetivos

- ✓ Conocer la estructura de una clase en el lenguaje Java.
- ✓ Comprender los mecanismos de encapsulación y ocultación de información.
- ✓ Analizar la situación del lenguaje Java con respecto a las funcionalidades demandadas para la programación orientada a objetos.
- ✓ Diferenciar el paso de información por referencia y por valor.
- ✓ Aprender a declarar y sobrecargar métodos y constructores.
- ✓ Conocer las librerías de uso más común en Java.

Mapa conceptual



Glosario

Deprecated. Obsoleto. Este término se usa comúnmente en la documentación de Java para señalar aquellos elementos del lenguaje que es mejor no seguir utilizando por causas relacionadas con eficiencia o seguridad.

Field. También llamado campo o variable miembro. Permite almacenar parte o la totalidad de la información correspondiente a un objeto.

Instancia. Cada uno de los objetos creados a partir de una clase.

Instanciación. Proceso de creación de objetos a partir de los constructores de una clase.

Librería. Archivo, o archivos, cuyo contenido es un conjunto de implementaciones que ofrecen una interfaz de comunicación bien conocida para realizar una tarea determinada.

Overloading. Técnica de programación que permite declarar varios métodos con el mismo identificador.

RAM. También llamada memoria de acceso aleatorio (*Random Access Memory*) es un tipo de memoria volátil en la cual se almacena la información del sistema operativo, los programas en ejecución y los datos manejados por estos.

Referencia. Dirección real de memoria RAM de una variable u objeto.

This. Palabra reservada que permite identificar los miembros de una clase frente a otros elementos cuando sus identificadores son coincidentes (situación de sobrecarga de nombres).

6.1. Concepto de clase

En los capítulos 2 y 4 se ofreció una visión general de lo que es una *clase*; en este capítulo se detallará ese concepto, así como la forma de implementar la clase y sus miembros.

Las clases generalizan entidades del mundo real y los objetos son ocurrencias concretas de esas entidades en un programa en ejecución. Dicho de otra manera y de forma coloquial, una clase es una plantilla para la creación de *objetos*, también llamados *instancias* o *instancias de clase*.

6.2. Estructura y miembros de una clase

La declaración de una clase se realiza indicando el nivel de visibilidad de la clase, su nombre, el nombre de la clase de la cual hereda (si no se indica, heredará implícitamente de la superclase *Object*) y las interfaces que pueda implementar. En el cuerpo de la clase se especifica qué datos se utilizarán para representar objetos y las operaciones que se podrán ejecutar sobre dichos objetos. A continuación, se muestra la sintaxis empleada para declarar una clase omitiendo la información de sus miembros (serán descritos más adelante):

```
[visibilidad] class Nombre [extends Superclase] [implements Interface1,
Interface2, ...] {
   //  declaraciones de campos
   ...
   //  declaraciones de constructores
   ...
   //  declaraciones de métodos
   ...
}
```

NOTAS: Los elementos indicados entre corchetes ([]) son opcionales. En un mismo archivo podrán declararse varias clases.

Desde el momento en que una clase se declara, se convierte en un tipo de dato y, por tanto, se podrán crear distintas variables de este tipo. En el apartado 6.2.3 (Constructores e instanciación de objetos) se explicará cómo crear objetos de diferentes clases.

Actividad propuesta 6.1

¿Es importante el orden en el que se declaren los miembros de una clase? ¿Por qué?

6.2.1. Campos

Los objetos son modelados a partir de las clases y tienen un conjunto de *atributos* que determinan su estado. Esos atributos en Java se representan dentro del cuerpo de la clase mediante *campos*, también llamados *fields* o *variables miembro*. Esos campos serán propios de cada objeto salvo que se declaren como *estáticos*. Es recomendable declarar los campos al comienzo de la definición de clase. Esta es la sintaxis empleada para declarar los campos de una clase:

```
[Visibilidad] [static] tipo campo1;
```

En la sintaxis de declaración del campo se incluye un elemento que permite establecer su visibilidad y que será explicado con detalle en el apartado 6.3 (Encapsulación y visibilidad). Independientemente del valor asignado, el campo siempre será visible desde todos los métodos de la clase.

Dado que también es posible representar el estado asociado a una clase, existen *campos estáticos*, también conocidos como *campos de clase* o *variables de clase*. El valor que se asigne a dicho campo será compartido por todos los objetos de la clase. Para declarar un campo como estático será necesario añadir la palabra reservada *static* en su declaración. En el siguiente ejemplo puede verse cómo declarar *variables miembro* (*campo2*) y *variables de clase* (*campo1*):

```
public class Clase {
    static int campo1;
    int campo2;
}
```

Sabías que...

Los *campos* pueden inicializarse en la *declaración*, en el método *constructor* o en *bloques de inicialización*.

Los *campos estáticos* pueden inicializarse en la *declaración*, en *bloques de inicialización estática* o en *métodos estáticos*.

Actividad propuesta 6.2

Si un campo estático es compartido por todos los objetos de una clase:

¿Se puede establecer y recuperar su valor a partir de cualquier objeto?
¿Se puede establecer y recuperar su valor en caso de que no existan instancias de la clase?
Justifica tus respuestas.

6.2.2. Métodos

Los métodos son conjuntos de instrucciones definidas en una clase cuyo fin es la realización de una tarea o la implementación de un algoritmo. Por este motivo generalmente los identificadores empleados suelen corresponderse con verbos que describen la tarea que llevan a cabo. Están disponibles para cada objeto de la clase en la que se define. Aunque lo normal es que utilicen los datos contenidos en la clase, es posible que no los necesite.

En cuanto a la sintaxis de declaración, la primera línea establecerá la *visibilidad*, *tipo de método*, *tipos de retorno*, *identificador* y *parámetros* de entrada. El cuerpo del método vendrá encerrado por llaves ({}) y contendrá su código, así como la declaración de variables locales. La sintaxis que se ha de emplear en la declaración de un método se describe a continuación:

```
[visibilidad] [static] tipo nombre ([tipo parám.1] ... [,tipo parám.N])
{
    // declaraciones locales al método
    ...
    [return valor]
}
```

En la sintaxis de declaración del método se incluye un elemento que permite establecer su visibilidad y que será explicado con detalle en el apartado 6.3 (Encapsulación y visibilidad). Independientemente del valor asignado, el método será visible desde todos los métodos de la clase.

Si el identificador de una variable local al método coincide con una variable miembro de la misma clase (*campo*), esta queda oculta al código del método. Para acceder a ella será necesario emplear la palabra reservada *this*, que únicamente puede utilizarse desde un método invocado por una instancia de clase. A continuación puede verse un ejemplo:

```
public class Ambitos {
    private int a = 22;
    public void Metodo (){
        int a = 33;
        System.out.println("Variable local a = " + a);
        System.out.println("Campo a = " + this.a);
    }
    public static void main(String[] args) {
        Ambitos ejemplo = new Ambitos();
        ejemplo.Metodo ();
    }
}
```

```
run:
Variable local a = 33
Campo a = 22
```


De igual manera que se pueden declarar campos estáticos, también se pueden declarar métodos estáticos. Para ello es necesario incluir la palabra reservada *static* en la declaración. Este tipo de métodos podrán ser accedidos a través del identificador de clase y sin necesidad de disponer de un objeto, aunque también están accesibles desde las instancias de clase.

A) Argumentos y parámetros

La declaración de un método podrá establecer tantos parámetros como sea necesario. Cada parámetro se declarará precedido de su tipo y vendrán separados por comas. Para invocar un método será necesario incluir en la llamada tantos argumentos como parámetros hayan sido especificados en su declaración y anteponer el nombre del objeto sobre el cual ejecutarlo. Si se trata de un método estático, se podrá utilizar el nombre del objeto o bien el de la clase.

Si el identificador de un parámetro del método coincide con una variable miembro de la misma clase (*campo*), esta queda oculta al código del método. Para acceder a ella será necesario emplear la palabra reservada *this*. A continuación puede verse un ejemplo:

```
public class Ambitos {
    private int a = 22;
    public void Metodo (int a){
        System.out.println("Parámetro a = " + a);
        System.out.println("Campo a = " + this.a);
    }
    public static void main(String[] args) {
        Ambitos ejemplo = new Ambitos();
        ejemplo.Metodo (11);
    }
}
```

```
run:
Parámetro a = 11
Campo a = 22
```

Paso por valor *versus* paso por referencia

En programación siempre se hace distinción entre el paso de argumentos a un método por valor o por referencia:

1. Si se pasan *por valor*, cuando se realiza la invocación a un método este recibe una copia del valor original y, por tanto, cualquier cambio sobre esta copia en el método no afecta al dato original.
2. Si se pasan *por referencia*, cuando se realiza la invocación a un método este recibe la dirección de memoria real del dato original y, por tanto, cualquier cambio desde el método sobre el dato afecta al dato original.

En Java, concretamente, siempre se realiza el paso *por valor* de todas las variables independientemente de que el tipo de dato pasado sea un *tipo primitivo* o un *objeto*. Pero para entender bien cómo trabaja Java es necesario conocer cómo gestiona la memoria:

1. Las *variables de tipo primitivo* se almacenan en la memoria *Stack*. Si se está pasando un dato de este tipo, lo que se pasa es una copia que se crea en el *Stack* de los bits que representan el valor original.
2. Los *objetos*, sin embargo, se almacenan en dos partes de la memoria:

   a) Por un lado, los datos del objeto (*campos*) se almacenan en la memoria *Heap*.
   b) Por otro lado, se almacena la dirección de memoria *Heap* en la que están los datos en la memoria *Stack* a modo de referencia o puntero.

Si se pasa un dato de este tipo, lo que se pasa es una copia que se crea en el *Stack* de los bits que representan la referencia al objeto (dirección del dato en la memoria *Heap*). Puesto que las dos variables (copia y original) que ahora residen en el *Stack* apuntan a la misma dirección de memoria del *Heap*, un cambio en los atributos de ese objeto apuntado desde la copia afectará al valor original.

Actividad propuesta 6.3

Averigua el modificador que impide que se cambie el valor de un parámetro dentro del código de un método.

B) Valores de retorno

Los métodos Java pueden devolver un valor como resultado. Si devuelve un valor, es necesario indicar el *tipo* del dato a devolver, que podrá ser un tipo primitivo o una clase, así como incluir la instrucción *return* seguida del valor a devolver dentro del cuerpo del método:

```
// método que devuelve la suma de dos números que recibe como parámetros
public int suma (int n1, int n2){
   return (n1 + n2);
}
```

Si, por el contrario, el método no devuelve ningún valor, como tipo en la declaración se indicará *void*:

```
// método que no devuelve ningún resultado ni recibe parámetros
public void holaMundo (){
    System.err.print("Hola Mundo !!!");
}
```

Sabías que...

Aunque lo habitual es colocar la instrucción *return* justo al final de cada método que devuelva un valor, en realidad puede ir delante de otras instrucciones. En caso de ejecutarse, provocará un salto incondicional que forzará la salida anticipada del método.

C) Sobrecarga de métodos (Overloading)

Java permite que una clase declare más de un método con el mismo identificador, pero para ello debe declarar diferentes parámetros (tanto en número como en el tipo de estos). Esta característica se denomina *sobrecarga de métodos* u *overloading*. Los métodos sobrecargados son diferenciados según el número y el tipo de los argumentos pasados en la invocación. A continuación se muestra un ejemplo de sobrecarga del método *suma*:

```
public int suma (int n1, int n2) {
   return (n1 + n2);
}
public int suma (int n1, int n2, int n3) {
   return (n1 + n2 + n3);
}
public double suma (double n1, double n2) {
   return (n1 + n2);
}
```

6.2.3. Constructores e instanciación de objetos

Toda clase dispone, aunque no se declare, de un método constructor que se tendrá que invocar sin argumentos. Además de este, se podrán implementar tantos métodos constructores como se desee. La sintaxis de un método constructor es similar a la de un método convencional con la diferencia de que no tiene nombre y que el tipo retornado es el mismo que el de la clase en la cual se declara. Para invocar al constructor e instanciar un objeto de clase, se debe hacer una llamada normal al método constructor pero precedido del operador *new*.

En el siguiente ejemplo puede verse cómo se declaran y utilizan métodos, campos y constructores:

```
class EjemploMetodos {
   public static void main(String[] args) {
        Saludo s1 = new Saludo(null);
        Saludo s2 = new Saludo("Mario");
        s1.saludar("Luis");
        s2.saludar();
        Utilidades.saludar("Ana");
        s2.comienzoSaludo = "Adiós ";
```

```
Salida:

Hola Luis
Hola Mario
Hola Ana
Adiós Luis
Adiós Mario
Adiós Ana
```

```
        s1.saludar("Luis");
        s1.saludar(s2.getNombre());
        Utilidades.saludar("Ana");
 }
}
class Saludo {
   static String comienzoSaludo = "Hola ";
   String nombre;
   Saludo (String nombre){
        this.nombre = nombre;
 }
 void saludar(){
      System.out.println(comienzoSaludo + nombre);
 }
 void saludar(String nombre){
      System.out.println(comienzoSaludo + nombre);
 }
 String getNombre(){
      return nombre;
 }
}
class Utilidades {
 static void saludar(String nombre){
      System.out.println(Saludo.comienzoSaludo + nombre);
 }
}
```

6.2.4. Destructores y liberador de memoria

Un *método destructor* se encarga de destruir objetos y liberar los recursos que tuviese asignados. En Java no existe este tipo de métodos, a diferencia de otros lenguajes como C++ o Delphi que sí lo implementan y hacen necesario su uso para evitar pérdidas de memoria durante la ejecución de los programas.

Pero no tener un método destructor no implica que no se realice liberación de memoria. Java dispone de un mecanismo denominado recolector de basura que se encarga de esta tarea y que hace que el desarrollo pueda ser más fluido al no tener el programador que preocuparse de realizarla (en el apartado 3.3.6 se explica su funcionamiento). Sin embargo, el recolector de basura no libera la memoria de forma inmediata.

Si se desea ejecutar el recolector de basura de forma inmediata, se puede invocar al método *System.gc()*. Este método llamará al método *finalize()* de la clase base *Object* para cada objeto que sea necesario liberar. Para ejecutar una tarea concreta justo antes de que el objeto sea destruido, se puede sobrescribir el método *finalize* de la clase base indicando las tareas a realizar.

A continuación, se muestra un ejemplo de uso del liberador de memoria:


```
class RecolectorBasura {
 @Override
 public void finalize() {
      System.out.println("Limpiando");
      // Tareas de limpieza y liberación
      System.out.println("Limpieza finalizada ");
 }
 public static void main(String[] args) {
      RecolectorBasura rc = new RecolectorBasura();
      rc = null;
      System.gc();
      System.out.println("Fin aplicación");
   }
}
```

```
Limpiando
Limpieza finalizada
Fin aplicación
```

Para saber más

Los mecanismos de finalización expuestos en este apartado pueden generar problemas de rendimiento y no garantizan que su ejecución se produzca de forma inmediata. Su uso está totalmente desaconsejado (la versión 9 del JDK ha marcado el método *finalize()* como *Deprecated*).

6.2.5. Propiedades

Tal como se explicó en el capítulo 2, una *propiedad* no tiene asociada una reserva de espacio de memoria para su almacenamiento, sino que accede al área de almacenamiento a través del campo definido previamente o bien a través de un método creado para tal efecto. El lenguaje de programación Java no ofrece mecanismos para implementarlas.

Una débil aproximación a las propiedades que otros lenguajes sí implementan (C#, Delphi, VB.NET…) es la declaración de métodos *get* y *set* para la obtención y establecimiento de los valores de los campos declarados con visibilidad *private*. A continuación se muestra un ejemplo típico de declaración de propiedades para una clase:

```
public class Animal {
   private String especie;
   public String getEspecie (){ return especie; }
   public void setEspecie (String e) { especie = e; }
}
```

Actividad propuesta 6.4

Propón el código de la clase Producto empleada para generar la siguiente salida a partir del código main suministrado:

```
run:

Rebaja actual: 0.0%.
Precio p1: 10.0
Precio p2: 30.0

Rebaja actual: 25.0%.
Precio p1 primeras rebajas: 7.5
Precio p2 primeras rebajas: 22.5

Rebaja actual: 50.0%.
Precio p1 segundas rebajas: 5.0
Precio p2 segundas rebajas: 15.0
```

```
public static void main(String[] args) {
    // creando 2 productos
    Producto p1 = new Producto(10);
    Producto p2 = new Producto(30);
    Producto.infoRebajaActual();
    System.out.println("Precio p1: " + p1.getPrecioFinal());
    System.out.println("Precio p2: " + p2.getPrecioFinal());
    // primeras rebajas y se decide bajar todos los precios un 10 %
    p2.rebaja = 25;
    p1.infoRebajaActual();
    System.out.println("Precio p1 primeras rebajas: " + p1.getPrecioFinal());
    System.out.println("Precio p2 primeras rebajas: " + p2.getPrecioFinal());
    // segundas rebajas y se decide bajar todos los precios a la mitad (50 %)
    Producto.rebaja = 50;
    p2.infoRebajaActual();
    System.out.println("Precio p1 segundas rebajas: " + p1.getPrecioFinal());
    System.out.println("Precio p2 segundas rebajas: " + p2.getPrecioFinal());
}
```

6.3. Encapsulación y visibilidad

El lenguaje de programación Java implementa el concepto de *encapsulación* mediante los modificadores de visibilidad que permiten establecer a los elementos contenedores (*paquete* y *clase*) y a los elementos en ellos contenidos (miembros *campo* y *método*). De esta manera es posible ocultar los campos de un objeto para que únicamente sea posible acceder a ellos mediante los métodos definidos para tal propósito. Esta encapsulación facilita la creación de *librerías* con métodos e interfaces bien definidos, favoreciendo la reutilización del código.

Los modificadores de visibilidad determinan si un miembro de una clase puede ser accedido por elementos de otras clases. En Java existen dos niveles de control de acceso sobre los que establecer modificadores de visibilidad: *nivel superior (clases)* y *nivel de miembro*.

Recuerda

✓ Para escribir un código eficiente y seguro se recomienda evitar el uso de *public* (salvo en *constantes*, método *main* o si es estrictamente necesario) y utilizar los modificadores más restrictivos posibles.

Cuadro 6.1
Modificadores de visibilidad

Nivel superior (clases)	
`public`	clase visible para todas las clases
`package-private` (sin modificador definido)	clase visible para clases del mismo paquete

Nivel de miembro				
Modificador miembro	Visible desde			
	Misma clase	Clases y subclases del mismo paquete	Subclases en otro paquete	Otras clases
`public`	Sí	Sí	Sí	Sí
`protected`	Sí	Sí	Sí	No
`package-private` (sin modificador definido)	Sí	Sí	No	No
`private`	Sí	No	No	No

6.4. Librerías y paquetes de clases

Como se habrá podido comprobar, para el desarrollo de un programa simple no es necesario realizar la importación de ninguna librería. Eso se debe a que, por defecto, ya se importa la librería *java.lang* que contiene clases esenciales para el lenguaje Java. Sin embargo, cuando el programa va creciendo en complejidad y necesitando de elementos más sofisticados, se hace indispensable el uso de nuevas librerías.

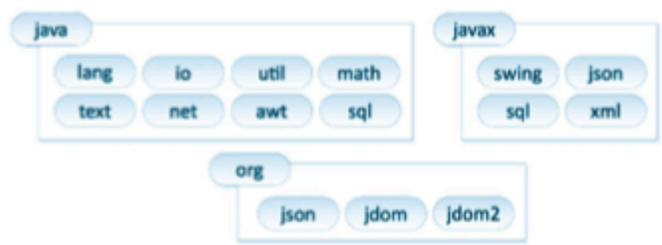


Figura 6.1
Librerías de uso común en Java.

No solamente se podrán importar librerías del entorno Java que están disponibles desde su entorno de desarrollo, sino que también podrá cargar librerías desarrolladas por el propio usuario o terceros y que hayan sido empaquetadas para ofrecer alguna funcionalidad.

Resumen

- Las clases y paquetes favorecen la encapsulación ayudados por los modificadores de visibilidad.
- En Java:
  - Las clases se componen de *campos, métodos* y *constructores*.
  - Los campos y métodos pueden ser estáticos.
  - Existen diferentes niveles de visibilidad.
  - No se implementan *properties*.
  - El programador no se debe preocupar por la gestión de memoria (se desaconseja la interacción manual con el recolector de basura).
  - El paso de datos a los métodos se hace siempre por valor.
  - Hay muchas librerías disponibles que facilitan la edición de código.

Ejercicios propuestos

1. Describe la diferencia existente entre paso de datos *por valor* y *por referencia*. Ilustra la explicación con un gráfico o diagrama.
2. ¿Qué diferencias existen entre una variable *de clase* y una variable *miembro*?
3. ¿Qué alternativa sugerirías implementar ante la imposibilidad de utilizar propiedades?
4. ¿Pueden los datos de un objeto que se pasa como argumento a un método variar cuando haya finalizado la ejecución del método? Razona tu respuesta.
5. ¿En qué consiste la sobrecarga de métodos? Pon un ejemplo.
6. Realiza un estudio sobre las librerías expuestas. Alguna puede que no esté disponible en tu distribución, ¿por qué?

Supuestos prácticos

1. Implementa un programa que genere dos números aleatorios, los muestre por pantalla y muestre también el valor del mayor sin usar ninguna sentencia de control (utiliza la función *nextInt* del objeto *Random* contenido en la *librería java.util* y la función *Math.Max* de la librería ya incluida *java.lang*).
2. Implementa la clase *viajero* que sea capaz de almacenar información sobre DNI, nombre, apellidos, nacionalidad y dirección.
3. Añade a la declaración de la clase *viajero* los métodos apropiados para lectura y escritura de los campos y establece los niveles de visibilidad adecuados a los miembros.

4. Incluye la clase en un paquete denominado *viajero*, que a su vez esté contenido dentro del paquete *persona*.
5. Declara el método constructor necesario para crear un objeto de la clase *viajero* pasando como argumento los datos de todos los campos, así como otro constructor que inicialice los valores a un valor por defecto y que no reciba ningún parámetro.
6. Incluye en la clase un campo estático que almacene el número de instancias creadas de la clase (deberá incrementarse en cada ejecución del método constructor).
7. Añade a la clase *viajero* el método *getInfo* que retorne información del viajero. Deberá ser declarado con el parámetro *Formato*. Este parámetro admitirá el valor *"pausado"* en cuyo caso irá mostrando por pantalla la información de todos los campos y esperará a recibir un *intro* por parte del usuario para seguir mostrando los siguientes, o *"normal"*, en cuyo caso se mostrará toda la información en un único paso.

Actividades de autoevaluación

1. ¿Es posible invocar directamente al método *finalize*()?
   - ☐ *a)* Sí.
   - ☐ *b)* Sí, pero únicamente desde la clase pública del paquete.
   - ☐ *c)* No, se llama directamente al ejecutar el método *System.gc*().
   - ☐ *d)* Todas las opciones anteriores son incorrectas.
2. Si no se establece un modificador de visibilidad a un método, por defecto será:
   - ☐ *a)* *Private*.
   - ☐ *b)* *Package-private*.
   - ☐ *c)* *Protected*.
   - ☐ *d)* Todas las opciones anteriores son incorrectas.
3. ¿Cuál de las siguientes afirmaciones es falsa?
   - ☐ *a)* En Java los parámetros no se pueden pasar por referencia.
   - ☐ *b)* Si un parámetro se pasa por referencia a un método, este recibe la dirección de memoria real del dato original y, por tanto, cualquier cambio desde el método sobre el dato afecta al dato original.
   - ☐ *c)* Las opciones *a)* y *b)* son correctas.
   - ☐ *d)* Todas las afirmaciones anteriores son falsas.

4. Un método estático se puede invocar:

- ☐ *a)* Sin necesidad de tener creado un objeto de la clase en la que fue declarado.
- ☐ *b)* Desde un objeto de la clase en la que fue declarado.
- ☐ *c)* Las opciones *a)* y *b)* son correctas.
- ☐ *d)* Las opciones *a)* y *b)* son incorrectas.

5. Si se necesita utilizar el tipo Date de la librería java.util:

- ☐ *a)* Se puede hacer *import java.util.Date;* y ya estará accesible el tipo *Date*.
- ☐ *b)* Se puede hacer *import java.util.*;* y ya estará accesible el tipo *Date*.
- ☐ *c)* Se puede acceder directamente al tipo *java.util.Date*.
- ☐ *d)* Todas las opciones son correctas.

6. Un campo puede inicializarse:

- ☐ *a)* En la declaración.
- ☐ *b)* En el método constructor.
- ☐ *c)* En bloques de inicialización.
- ☐ *d)* Todas las opciones anteriores son correctas.

7. Si un miembro se declara con visibilidad *package-private*:

- ☐ *a)* Podrá verse desde la misma clase, desde subclases de la misma clase en la que se define y desde clases del mismo paquete.
- ☐ *b)* Podrá verse desde la misma clase, pero no desde clases del mismo paquete.
- ☐ *c)* Podrá verse desde la misma clase, pero no desde las subclases.
- ☐ *d)* Todas las opciones anteriores son incorrectas.

8. ¿Qué nivel de visibilidad establecerías para que una clase fuese visible para clases del mismo paquete?

- ☐ *a)* *Package.*
- ☐ *b)* *Package-private.*
- ☐ *c)* *Public.*
- ☐ *d)* *Protected.*

9. Teniendo en cuenta la sintaxis de declaración de una clase, ¿cuál de las siguientes es la correcta?

- ☐ *a)* *[visibilidad] class Nombre [extends Superclase] [implements Interface1, Interface2, …]*
- ☐ *b)* *visibilidad class Nombre [extends Superclase] [implements Interface1, Interface2, …]*
- ☐ *c)* *[visibilidad] class Nombre extends Superclase [implements Interface1, Interface2, …]*
- ☐ *d)* *class [visibilidad] Nombre [extends Superclase] [implements Interface1, Interface2, …]*

10. Identifica cuál de las siguientes afirmaciones es falsa:

- ☐ *a)* Los objetos se almacenan en dos partes de memoria (*Heap* y *Stack*).
- ☐ *b)* Los parámetros declarados en un método van precedidos de su tipo y separados por comas.
- ☐ *c)* Las variables de tipo primitivo se almacenan en la memoria *Heap*.
- ☐ *d)* Si un argumento se pasa por referencia, el método recibe la dirección de memoria real del dato original.

SOLUCIONES:

1.	a	b	**c**	d	**5.**	a	b	c	**d**						
2.	a	**b**	c	d	**6.**	a	b	c	**d**	**9.**	**a**	b	c	d	
3.	a	b	c	**d**	**7.**	**a**	b	c	d	**10.**	a	b	**c**	d	
4.	a	b	**c**	d	**8.**	a	**b**	c	d						

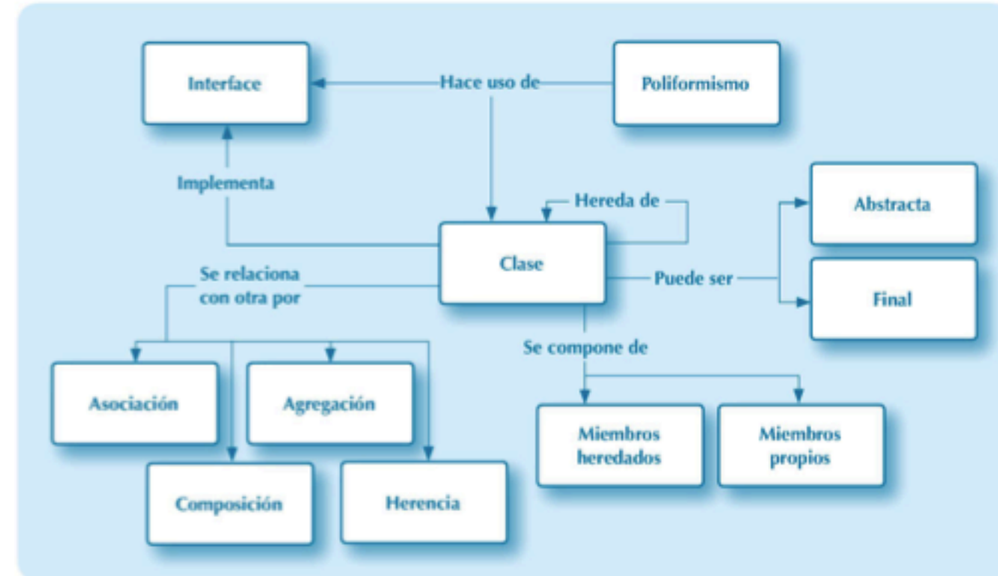
7

Clases avanzadas y utilización de objetos

Objetivos

- ✓ Conocer los diferentes tipos de relaciones que se pueden dar en un escenario y aprender a reflejarlos en un programa.
- ✓ Comprender el comportamiento de los diferentes miembros en la herencia.
- ✓ Analizar las diferentes posibilidades para la implementación de herencia múltiple en Java.
- ✓ Diferenciar la sobreescritura de la sobrecarga de métodos.
- ✓ Aprender a acceder a los miembros de las superclases.
- ✓ Conocer el polimorfismo y aprender a manejarlo.

Mapa conceptual



Glosario

Agregación. Relación de composición débil en la que los objetos componentes tienen entidad suficiente como para existir en el sistema sin necesidad de que lo haga el objeto compuesto.

Asociación. Relación existente entre objetos de esta u otras clases que interaccionan entre sí.

Clase abstracta. Clase que obvia determinados detalles y que por tanto no es completamente funcional. Se emplea con la finalidad de implementar relaciones de herencia y polimorfismo.

Composición. Relación entre un objeto compuesto y objetos componentes, en la que los componentes no existen si no existe el compuesto.

Herencia. La herencia es la relación existente entre dos clases, tal que una clase toma (hereda) campos y métodos de las clases base que puede modificar.

Object. Nombre de la clase base de todas las clases en Java. Si una clase no extiende otra clase de forma explícita, entonces es una clase hija directa de *Object*. Si por el contrario extiende otra clase, entonces sí que será una clase hija indirecta de *Object*.

Parametrizado. Se dice que un método o constructor es parametrizado cuando recibe parámetros a través de los argumentos de la llamada.

Polimorfo. Se dice que algo es polimorfo cuando puede tener varias formas.

Pseudo. Que no es del todo real o que aparenta ser algo que en realidad no es.

Relación transitiva. Se dice que entre los elementos de un conjunto existe una relación transitiva cuando se garantiza que, si un elemento tiene una relación con un segundo elemento y este último con un tercero, entonces el primero tiene la misma relación con el tercero.

UML. Abreviatura de *Unified Modeling Language,* es un lenguaje de modelado estandarizado consistente en un conjunto integrado de diagramas que sirven de ayuda al desarrollo de software.

7.1. Asociación, agregación y composición de clases

En el capítulo anterior se detalló el concepto de *clase*, especificando todos los miembros que puede contener y cómo debe declararse. Sin embargo, no se ha mencionado en ningún momento las relaciones que pueden existir entre elementos de diferentes clases. Según lo expuesto en el capítulo 2, los *campos* de una clase podrán pertenecer a clases ya declaradas formando relaciones de:

1. *Asociación:* los objetos de una o varias clases interactúan entre sí y cada uno puede existir por sí mismo sin depender del otro. Este es un programa ejemplo que utiliza *asociación*:

```
class Universidad  {
      private String nombre;
      Universidad (String name) {
           this.nombre = name;
 }
 public String getNombreUniversidad() {
      return this.nombre;
 }
}
```

```
class Alumno {
     private String nombre;
     Alumno(String nombre) {
          this.nombre = nombre;
      }
      public String getNombreAlumno() {
           return this.nombre;
      }
}
```

Asociación

La relación existente entre los objetos Universidad y Alumno se da en el programa principal, de manera que ambos pueden existir en el sistema sin depender uno del otro.

La palabra reservada this representa al objeto desde el que se está ejecutando el código. De esta manera permite acceder a sus campos cuando existen parámetros con el mismo nombre.


```
// Asociación entre ambas clases en método principal
class Asociacion  {
      public static void main (String[] args)  {
           Universidad u;
           u = new Universidad ("Universidad Oxford");
           Alumno a;
           a = new Alumno ("Flora");
           System.out.println(a.getNombreAlumno() +
                " es alumn@ de " +
                u.getNombreUniversidad());
 }
}
```

2. *Agregación:* es una *asociación* entre un objeto compuesto y objetos componentes, en la que los componentes pueden existir por sí mismos sin necesidad del compuesto. Este es un programa de ejemplo que utiliza *agregación:*

```
class Persona {
      private Persona padre;
      Persona (Persona padre) {
           this.padre = padre;
 }
}
```

Agregación

Un objeto persona podrá existir aunque no exista el objeto persona que represente a su padre.

3. *Composición:* es una *asociación* entre un objeto compuesto y objetos componentes, en la que los componentes no existen si no existe el compuesto. Este es un programa de ejemplo que utiliza *composición:*

```
class Punto {
      private int x;
      private int y;
      public Punto(int coord_X, int coord_Y) {
           x = coord_X;
           y = coord_Y;
 }
 public Punto() {
      x=0;
      y=0;
 }
}
```

```
class Circunferencia {
      private Punto centro = new Punto ();
      private int radio;
      public Circunferencia(int radio){
           this.radio = radio;
      }
}
```

Composición

No tiene sentido la existencia del objeto punto que formará el centro de la circunferencia en ausencia de esta. Si el objeto circunferencia deja de existir, también dejará de existir el objeto punto que representa su centro.

Actividades propuestas

7.1. Propón escenarios ejemplo de los tres tipos de relación expuestos.

7.2. En los ejemplos mostrados no se tratan conceptos como *cardinalidad* o *multiplicidad* para facilitar la exposición. Investiga sobre ellos y aporta ejemplos.

7.2. Herencia

La *herencia* permitirá declarar una nueva clase (*derivada*, *extendida*, *hija* o *subclase*) a partir de otra existente (*ancestra*, *base*, *padre* o *superclase*). Gracias a este mecanismo de herencia todos los miembros de la *superclase* son transferidos a la *subclase*, excepto los constructores.

En Java únicamente se dispone de herencia simple. Esta es una *relación transitiva* mediante la cual, si una clase C hereda de la clase B, y la clase B hereda de la clase A, se puede decir que C también hereda de la clase A. Por este motivo todas las clases, independientemente de que hereden directamente de alguna clase o no, siempre heredarán de la superclase *Object*, a excepción de la misma clase *Object* que no hereda de ninguna otra clase.

Para declarar las relaciones de herencia se deberá añadir la palabra reservada *extends* seguida del nombre de la superclase desde la que se quiere heredar justo después del nombre de la clase. A continuación, se muestra un ejemplo de cómo implementar la herencia:

```
class subclase extends superclase {
    ...
}
```

7.3. Constructores y herencia

Cuando se trata de herencia, los constructores se comportan de manera muy diferente al resto de elementos. A continuación, se describen las particularidades de estos métodos:

1. Al contrario de lo que ocurre con los métodos y campos que son transferidos de una *superclase* a la *subclase*, los métodos constructores no se heredan. Por lo tanto, cada *subclase* tiene sus propios métodos constructores.
2. Aunque no se declare, cada clase tiene, de forma implícita, un constructor por defecto (sin parámetros ni código). Pero si se declara cualquier constructor, el constructor por defecto no se creará. Tal como se muestra en el siguiente ejemplo, es posible declarar tantos constructores parametrizados diferentes como sea necesario:

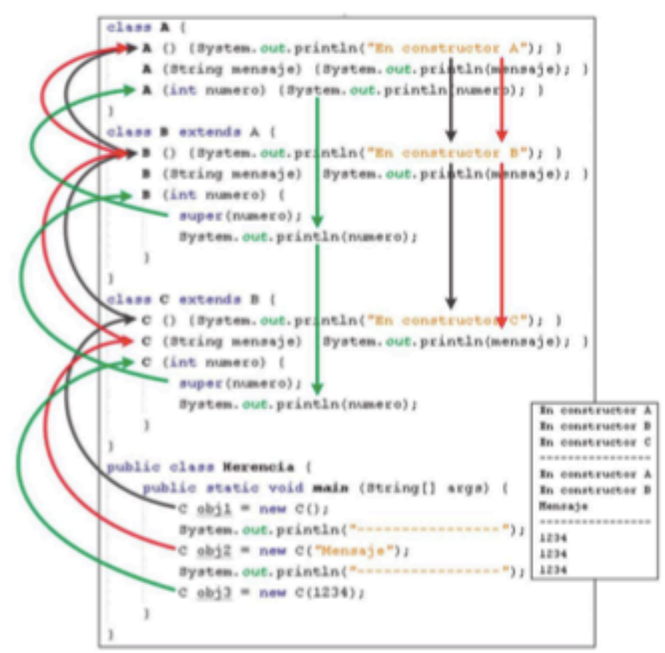
```
class Clase {
    Clase () {}  // sobrescribe el constructor por defecto
    Clase (String parametro1) {}
    Clase (int parametro1, int parametro2) {}
    // ... declaraciones de campos
}
class Superclase {}
class Subclase extends Superclase {}
public class Herencia {
    public static void main (String[] args) {
        Superclase sup = new Superclase ();
        Subclase sub = new Subclase ();
        Clase cla = new Clase ();
    }
}
```

3. Cuando se crea un objeto de una *clase*, no se invoca únicamente a su constructor, sino que se ejecuta el de todas las *superclases*. Se recorrerán todas las clases de la jerarquía hasta llegar a la clase padre, formando una pila de llamadas y buscando el constructor correspondiente:

   a) Si se invoca al constructor con algún argumento, se buscará el método constructor cuyos parámetros "encajen" con dicha llamada. En caso contrario se buscará el método constructor que no tenga parámetros, el cual podrá ser el constructor por defecto (sin parámetros ni código) o bien uno declarado por el programador (también sin parámetros).

   b) Si se desea forzar la llamada a un constructor concreto, será necesario invocarlo con la sentencia *super* como primera instrucción del constructor actual y pasarle los argumentos que espere recibir.

A continuación, se muestra un ejemplo de declaración y uso de constructores, en el que se puede comprobar cómo se realizan las diferentes llamadas en función del tipo de constructor invocado:



7.4. Acceso a campos de la superclase

En una relación de herencia, cuando un campo tiene el mismo identificador que un campo de la clase base, este último se oculta (incluso cuando los tipos de ambos no son coincidentes). El campo ocultado podrá ser accedido:

1. Desde instancias de la *subclase*:

   a) Mediante la palabra reservada *super* (únicamente se puede subir un nivel en la jerarquía).
   b) Mediante *cast* explícito a la superclase cuyo campo se ocultó.

2. Desde fuera de la *subclase* mediante un *cast* explícito a la superclase cuyo campo se ocultó.

En el siguiente ejemplo se muestra cómo realizar el acceso a los diferentes campos de una clase cuyo identificador coincide con el de su superclase, así como a los campos de las superclases de la jerarquía:

```
package herenciasobrescritura;
class A {
    String c1 = "rojo";
    String c2 = "negro";
}
class B extends A {
    int c1 = 1;
}
class C extends B {
    char c1 = 'Z';
    void verCampos(){
        System.out.println("----- Acceso desde clase C -----");
        System.out.println("Campo c2 de C = " + c2);
        System.out.println("Campo c1 de C = " + c1);
        System.out.println("Campo c1 de C = " + this.c1);
        System.out.println("Campo c1 oculto de B = " + super.c1);
        System.out.println("Campo c1 oculto de B = " + ((B)this).c1);
        System.out.println("Campo c1 oculto de A = " + ((A)this).c1);
    }
}
public class Herenciasobrescritura {
    public static void main (String[] args) {
        C obj = new C ();
        // Acceso desde el exterior de la clase C (método main)
        System.out.println("----- Acceso desde exterior -----");
        System.out.println("Campo c2 de C = " + obj.c2);
        System.out.println("Campo c1 de C = " + obj.c1);
        System.out.println("Campo c1 oculto de B = " + ((B)obj).c1);
        System.out.println("Campo c1 oculto de A = " + ((A)obj).c1);
        // Acceso desde la misma clase C
        obj.verCampos();
    }
}
```

```
----- Acceso desde exterior -----
Campo c2 de C = negro
Campo c1 de C = Z
Campo c1 oculto de B = 1
Campo c1 oculto de A = rojo
----- Acceso desde clase C -----
Campo c2 de C = negro
Campo c1 de C = Z
Campo c1 de C = Z
Campo c1 oculto de B = 1
Campo c1 oculto de B = 1
Campo c1 oculto de A = rojo
```

7.5. Acceso a métodos de la superclase

De la misma forma que se permite el acceso a los constructores y campos de la *superclase*, también se puede hacer uso de sus métodos desde las *subclases*, siempre y cuando estos sean visibles.

Para ello es necesario hacer uso nuevamente de la sentencia *super*, pero en este caso no es necesario que sea la primera instrucción del bloque como en los constructores.

7.6. Sobreescritura de métodos (*Override*)

Una *subclase* heredará de su *superclase* los métodos que le sean accesibles, a menos que la *subclase* los sobrescriba, en cuyo caso ya no dispondrá de ellos. Para sobrescribir un método es necesario declararlo con las mismas características (*identificador*, *número* y *tipo de parámetros*) que el método de la superclase. Generalmente la sobreescritura de métodos sirve para agregar o modificar la funcionalidad del método heredado de la *superclase*. Cuando se sobrescribe un método, es recomendable marcarlo con *@Override*. Al sobrescribirse el método este se personaliza para los objetos de la clase, pero el método original seguirá también estando disponible para objetos de la *superclase* u otras *subclases* que no lo sobrescriban. A continuación, se muestra un programa ejemplo en el que se lleva a cabo la sobreescritura de varios métodos:

```
package herenciasobrescritura;
class A {
    public void metodoA() {System.out.println("metodoA en A"); }
    public void metodoSobrescrito1() {System.out.println("metodoSobrescrito1 en A"); }
    public void metodoSobrescrito2() {System.out.println("metodoSobrescrito2 en A"); }
}
class B extends A {
    public void metodoB() {System.out.println("metodoB en B"); }
    @Override
    public void metodoSobrescrito1() {System.out.println("metodoSobrescrito1 en B"); }
    @Override
    public void metodoSobrescrito2() {super.metodoSobrescrito2();}
}
class C extends B {
    public void metodoC() {System.out.println("metodoC en C"); }
    @Override
    public void metodoSobrescrito1() {System.out.println("metodoSobrescrito1 en C"); }
    @Override
    public void metodoSobrescrito2() {super.metodoSobrescrito2();}
}
public class Herenciasobrescritura {
    public static void main (String[] args) {
        C obj = new C ();
        obj.metodoA(); obj.metodoB(); obj.metodoC();
        System.out.println("-----------------------");
        obj.metodoSobrescrito1();
        ((B)obj).metodoSobrescrito1();
        ((A)obj).metodoSobrescrito1();
        System.out.println("-----------------------");
        obj.metodoSobrescrito2();
    }
}
```

```
metodoA en A
metodoB en B
metodoC en C
-----------------------
metodoSobrescrito1 en C
metodoSobrescrito1 en C
metodoSobrescrito1 en C
-----------------------
metodoSobrescrito2 en A
```

7.7. Clases y métodos abstractos

Imaginemos un escenario en el que no tenga sentido crear cierto tipo de objetos o en el que se desea restringir su creación, pero en el que, sin embargo, dichos objetos siguen siendo de utilidad para generalizar miembros que otros tipos de objetos heredarán. En ese caso estaremos ante un claro ejemplo de *clase abstracta*.

Una clase abstracta no puede instanciarse y, por tanto, no debe tener constructores. Sin embargo, sí que puede implementar alguno de sus métodos. Todas las subclases que hereden de una clase abstracta podrán sobrescribir los métodos que estén a su alcance.

Para declarar una clase como abstracta basta con añadir la palabra reservada *abstract* justo antes de la palabra reservada *class*.

7.7.1. Métodos abstractos

Una *clase abstracta* evita la instanciación de objetos de esta, pero no obliga a sobrescribir ningún método de los que declare a las *subclases* que de ella hereden. Sin embargo, es posible forzar a las *subclases* de una *clase abstracta* a sobrescribir ciertos métodos declarándolos como abstractos. De no hacerlo deberán ser declaradas a su vez como clases abstractas.

Un *método abstracto* se declara mediante la palabra reservada *abstract* al comienzo del mismo, no tendrá cuerpo (es decir, no hay bloque encerrado entre llaves y su signatura finaliza con un punto y coma ";") y únicamente puede estar declarado dentro de una *clase abstracta*. Es decir, si una clase va a contener un método abstracto, obligatoriamente tendrá que ser declarada como *clase abstracta*. En este ejemplo se puede observar cómo declarar clases y métodos abstractos:

```
abstract class ClaseAbstracta {
   abstract void metodoAbstracto();
}
abstract class SubclaseAbstracta extends ClaseAbstracta{}
class Subclase extends ClaseAbstracta {
 @Override
 public void metodoAbstracto() {
      System.out.println("Implementando método abstracto");
   }
}
public class HerenciaAbstractos {
   public static void main (String[] args) {
        new Subclase().metodoAbstracto();
 }
}
```

7.8. Clases y métodos finales

Una clase puede declararse como *final* para evitar que sea utilizada como superclase desde la cual heredar, bien porque se ha llegado a la conclusión de que su código es inmejorable o bien por motivos de seguridad. Para ello habrá que añadir la palabra *final* antes del modificador de visibilidad de la clase.

7.8.1. Métodos finales

La *subclase* podrá añadir métodos propios o bien sobrescribir los métodos heredados; esto último salvo que explícitamente se indique lo contrario con la palabra reservada *final* al comienzo del

encabezado del método, haciendo de esta manera al método totalmente inalterable. A continuación, se muestra un ejemplo de declaración de clases y métodos finales:

```
final public class ClaseFinal {
    String cad = "ejemplo";
    public void metodoEjemplo() {
        System.out.println(cad);
    }
}
cannot inherit from final ClaseFinal
----
(Alt-Enter shows hints)
class SubclaseFinal extends ClaseFinal()
```

```
class Clase {
    String cad = "ejemplo";
    final public void metodoFinal() {
        System.out.println(cad);
    }
}
class SubclaseFinal extends Clase {
metodoFinal() in SubclaseFinal cannot override metodoFinal() in Clase
 overridden method is final

Add @Override Annotation
----
(Alt-Enter shows hints)
    public void metodoFinal()()
}
```

Actividad propuesta 7.3

¿Existe alguna relación entre la etiqueta *@Override* y los métodos declarados como final?

7.9. Interfaces

Cuando se va a desarrollar un proyecto por varios programadores que no trabajarán de forma conjunta pero cuyos resultados tienen que unirse para obtener un resultado final, se hace imprescindible establecer un contrato que establezca cómo podrá comunicarse el código de cada uno de ellos sin necesidad de entrar en los detalles de cómo está programado. A este contrato se le denomina *interfaz*.

Cada interfaz:

- ✓ Es un tipo de referencia al igual que lo es una clase.
- ✓ Debe declararse en un fichero cuyo nombre coincida con el de la interfaz, salvo que se omita el modificador de visibilidad.
- ✓ Recoge en su declaración un conjunto de firmas de constantes y métodos (cabeceras) que serán implementados en las clases que hagan uso de ella. La clase que no implemente la *interfaz* al completo deberá ser declarada como *abstracta*.
- ✓ No puede ser instanciada.
- ✓ Puede servir de base para ser extendida por otras interfaces.

Toda clase que vaya a implementar los métodos declarados en una interfaz, deberá indicarlo en su declaración mediante la palabra reservada *implements* seguida del nombre de la interfaz. Además, una clase podrá implementar tantas interfaces como sea necesario, en cuyo caso deberá indicar el nombre de las mismas separados por comas.

A continuación, se muestra la sintaxis de declaración de una interfaz:

```
public interface Identificador [extends Interface1, Interface2, …] {
   //    declaración de constantes
 …
 //    firmas de los métodos
 [static] tipo nombre ([tipo parám.1] … [,tipo parám.N]);
}
```

En el siguiente programa ejemplo puede verse cómo se declara e implementa una interfaz:

Tren.java

```
package interfaces;
public class Tren implements Operable {
    private String id;
    private int velocidad;
    Tren (String id, int velocidad){
        this.id = id;
        this.velocidad = velocidad;
    }
    public String getIdVehiculo() {
        return ("Tren con ID: " + id);
    }
    public void acelerar(){
        velocidad++;
    }
    public void frenar(){
        velocidad--;
    }
    public int getVelocidad(){
        return velocidad;
    }
    public void resumirMetodosTren(){
        System.out.println("getVelocidad");
        Operable.resumirMetodosInterface();
    }
}
```

Operable.java

```
package interfaces;
public interface Operable {
    public String getIdVehiculo ();
    public void acelerar ();
    public void frenar ();
    default public void imprimirUbicacionGeografica (){
        System.out.println("Implementación por defecto en interface");
    }
    public static void resumirMetodosInterface(){
        System.out.println("getIdVehiculo, acelerar, frenar"
                + ", imprimirUbicacionGeografica, resumirMetodosInterface");
    }
}
```

Principal.java

```
package interfaces;
public class Principal {
    public static void main(String[] args) {
        Tren t = new Tren("013", 0);
        System.out.println(t.getIdVehiculo());
        System.out.println("Velocidad: " + t.getVelocidad());
        for (int i=0; i<20; i++)
            t.acelerar();
        System.out.println("Velocidad: " + t.getVelocidad());
        t.imprimirUbicacionGeografica();
        t.resumirMetodosTren();
    }
}
```

SALIDA

```
Tren con ID: 013
Velocidad: 0
Velocidad: 20
Implementación por defecto en interface
getVelocidad
getIdVehiculo, acelerar, frenar, imprimirUbicacionGeografica, resumirMetodosInterface
```

Sabías que…

A partir del JDK 8 una interfaz puede contener *métodos estáticos*, igual que una clase, y métodos *default*. Estos últimos estarán disponibles para todas las clases que implementen la interfaz a modo de método por defecto y podrán ser sobrescritos; además, permiten añadir funcionalidades sin necesidad de retocar las clases que ya implementasen la interfaz. Ambos tipos de método se pueden codificar en la interfaz y estar disponibles para todas las clases que implementen esa interfaz. En este apartado se ha declarado un ejemplo de cada tipo.

A partir del JDK 9 una interfaz puede contener *métodos privados*. La forma de representar en UML que una clase implementa una interfaz es mediante una línea discontinua.

Actividad propuesta 7.4

Busca información sobre los métodos privados de las interfaces que pueden implementarse con el JDK 9 y propón un ejemplo de uso.

7.9.1. Pseudoherencia múltiple

La tecnología Java no soporta herencia múltiple, pero sí la implementación de múltiples *interfaces*. Mediante el uso de interfaces, en casos muy concretos, se puede simular relaciones de herencia múltiple, aunque no siempre es posible puesto que las interfaces no soportan atributos. Por este motivo es muy importante elegir bien qué objeto modelar como *clase* y qué objeto modelar como *interfaz*.

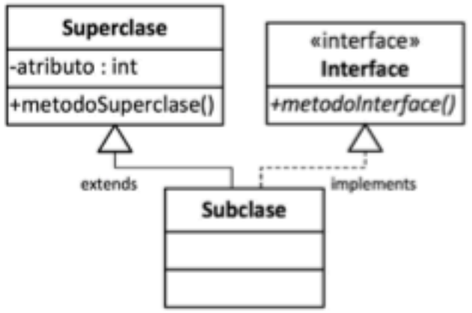


Figura 7.1
Ejemplo de herencia múltiple.

A continuación, se muestra la implementación del ejemplo de la figura 7.1 con dos variantes. En la variante A se realiza la implementación de la funcionalidad declarada en la interfaz desde la *subclase* (mecanismo tradicional). En la variante B se realiza la implementación desde la misma interfaz (método *default*).

```
package herenciamultiple;
public class Herenciamultiple {
    public static void main(String[]
   args) {
    Subclase obj = new Subclase();
        obj.metodoInterface();
        obj.metodoSuperclase();
   }
}
```

```
package herenciamultiple;
public class Superclase {
    int atributo;
    public void metodoSuperclase (){
            System.out.println("metodo
Superclase");
    }
}
```

SALIDA

```
metodoInterface
metodoSuperclase
```

A

```
package herenciamultiple;
public interface Interface {
    public void metodoInterface
    ();
}
```

```
package herenciamultiple;
public class Subclase extends Superclase
implements Interface{
    public void metodoInterface (){
        System.out.println("metodoInterface");};
}
```

B

```
package herenciamultiple;
public interface Interface {
    default public void
   metodoInterface (){
        System.out.println
        ("metodoInterface");
    }
}
```

```
package herenciamultiple;
public class Subclase extends Superclase
implements Interface{ }
```

7.10. Polimorfismo

El *polimorfismo* es la propiedad de los objetos que les permite adoptar diferentes formas en tiempo de ejecución, así como que un objeto de una clase derivada sea utilizado como un objeto de la clase base. Para que esta característica pueda ser utilizada es necesario que el lenguaje permita relajación de tipos.

7.10.1. Asignación polimorfa

Según el concepto de herencia, toda subclase es una especialización de la superclase. Dicho de otro modo, una superclase es una generalización de subclases. Por ende, cualquier objeto de una subclase es también un objeto de la superclase.

Partiendo de esta idea se puede entender mucho mejor el concepto *asignación polimorfa*, que es la asignación a una variable o campo declarados de tipo *clase*, siendo *clase* un término genérico, de un objeto perteneciente a una *subclase* de *clase*. Es decir, se podrán declarar variables o campos de un tipo *superclase* y posteriormente asignarle un objeto de alguna de las *subclases*. La variable o campo destino podrá ser declarada como *clase*, *clase abstracta* o *interfaz*.

7.10.2. Ejecución polimorfa

La *ejecución polimorfa* permite la invocación de métodos del mismo nombre sobre instancias de diferente clase sin reparar en ello. Garantiza que se ejecutará el método correspondiente a la *clase* real de un objeto, aunque el tipo con el que se presente sea otro. En el siguiente ejemplo se muestra cómo realizar *asignaciones polimorfas*, cómo hacer *ejecuciones polimorfas* y cómo acceder, mediante *casting*, al método que un objeto tiene pero que se oculta tras hacerse la asignación polimorfa:

A

```
package polimorfismo;
interface MedioTransporte {
    default public void getInfo () {
        System.out.println("MedioTransporte");
    }
}
class Aereo implements MedioTransporte {
    public void getInfo() { System.out.println("Aereo");}
}
```

B

```
package polimorfismo;
abstract class MedioTransporte {   // o class MedioTransporte {
    public void getInfo() {
        System.out.println("MedioTransporte");
    }
}
class Aereo extends MedioTransporte {
    public void getInfo() { System.out.println("Aereo");}
}
```



```
class Avion extends Aereo {
    private int numMotores;
    public Avion(int numMotores) { this.numMotores = numMotores; }
    public void getInfo() { System.out.println("Avion");}
    public void getNumMotores() {
        System.out.println("Tiene " + numMotores + " motores");
    }
}
class Helicoptero extends Aereo {
    public void getInfo() { System.out.println("Helicoptero");}
}
```

```
public class Polimorfismo {
    public static void main(String[] args) {
        MedioTransporte objMedioTransporte;
        objMedioTransporte = new Avion(4);
        objMedioTransporte.getInfo();
        System.out.println (objMedioTransporte.getClass().toString());
        ((Avion) objMedioTransporte).getNumMotores();
    }
}
```

```
SALIDA
Avion
class polimorfismo.Avion
Tiene 4 motores
```

Investiga

Investiga sobre el concepto *ligadura dinámica* y su relación con el concepto *polimorfismo*.

Resumen

- En Java es posible implementar las relaciones de composición, agregación, asociación y herencia.
- Java no soporta herencia múltiple, aunque puede simularla.
- Los constructores tienen un comportamiento especial y no son transferidos desde la superclase a las subclases.
- Las clases y métodos finales aportan seguridad ante posibles usos no deseados del código.

- Las clases y métodos abstractos pretenden forzar al programador a implementar parte del código
- La sobreescritura de métodos permite adecuar y modificar el comportamiento de los métodos heredados de la superclase.
- Una interfaz actúa a modo de contrato y determina un conjunto de métodos que las clases que la usan deben implementar.
- El polimorfismo permite almacenar objetos de subclases en variables o campos declarados como superclases o interfaces, así como ejecutar los métodos asociados a las subclases a través de esas referencias de tipo superclase.

Ejercicios propuestos

1. Describe la diferencia existente entre *métodos finales* y *métodos abstractos*.
2. Explica la diferencia existente entre *métodos sobrecargados* y *métodos sobrescritos*.
3. ¿Es posible acceder a los campos de una superclase desde una subclase? Justifica tu respuesta.
4. ¿Puede Java implementar la herencia múltiple? Razona tu respuesta.
5. ¿En qué consiste la ejecución polimorfa? Propón un ejemplo.
6. ¿Qué es una interfaz? Desarrolla tu respuesta.
7. Si se desea manejar información sobre, únicamente, las figuras geométricas punto, recta y circunferencia, ¿qué tipo de relación establecería entre ellas? Justifica tu respuesta.
8. Haz un resumen de las palabras reservadas que se emplean en el manejo de la herencia.
9. Si A es una clase de la cual hereda B y B es una clase de la cual hereda C, ¿es posible asignar a una variable declarada de tipo A un objeto de la clase C? ¿es posible asignar a una variable declarada de tipo C un objeto de la clase A? Justifica tus respuestas.

Supuestos prácticos

1. Codifica la jerarquía *ser vivo, animal* y *vegetal*, creando las estructuras que consideres necesario para almacenar de cada elemento, como mínimo, *nombre científico* y *nombre común*. Para los animales almacenar *peso* y *altura*. Para los vegetales únicamente *altura*. Cada ser vivo se alimentará de una manera diferente, por lo que deberá reflejar este aspecto de alguna forma.
2. Implementa el escenario propuesto en la figura 2.4 del capítulo 2 y añade las declaraciones de los métodos *get* y *set* para cada uno de los campos, así como constructores parametrizados a cada una de las clases.
3. Codifica la solución que hayas propuesto para el ejercicio propuesto 7 junto con un programa que muestre un menú que permita crear los tres objetos de las clases indicadas y mostrar la información de estas.
4. Supón que tu empresa se va a hacer cargo del desarrollo de un programa para el manejo de maquinaria de un fabricante. Las máquinas podrán ser de tipo industrial o bien de tipo doméstico. Independientemente de su tipo, desde el programa debe poderse manejar información relativa al número de serie de cada una, modelo y horas de funcionamiento. Todas las máquinas, como mínimo, deberán poderse encender y apagar, pero obviamente cada una lo hará de una forma concreta debido a sus características internas. Para toda la maquinaria industrial, además, deberá poderse conocer la potencia de consumo. Implementa el escenario propuesto.

Actividades de autoevaluación

1. Si la clase A hereda de la clase B:
   - ☐ *a)* Todos los campos de B son transferidos a la clase A.
   - ☐ *b)* Todos los métodos de B son transferidos a la clase A.
   - ☐ *c)* Las opciones *a)* y *b)* son correctas.
   - ☐ *d)* Todas las opciones anteriores son incorrectas.

2. ¿Cuál de las siguientes afirmaciones es verdadera?
   - ☐ *a)* En una relación de composición, el compuesto debe existir previamente al componente.
   - ☐ *b)* En una relación de composición, el componente debe existir previamente al compuesto.
   - ☐ *c)* En una relación de asociación, el compuesto debe dejar de existir si lo hace el componente.
   - ☐ *d)* Todas las opciones anteriores son incorrectas.

3 ¿Puede una variable declarada de un tipo *clase abstracta* contener un objeto de otra clase?
   - ☐ *a)* No.
   - ☐ *b)* Sí, pero únicamente si el objeto es de una subclase que implementa completamente la clase abstracta.
   - ☐ *c)* Sí, puesto que al ser abstracta puede contener referencias a objetos de cualquier tipo.
   - ☐ *d)* Sí, pero únicamente si el objeto es de tipo *Object*.

4. Un método abstracto:

- ☐ *a)* No puede ser codificado en la clase en la que se declara.
- ☐ *b)* Obliga a que la clase en la que se declara sea también abstracta.
- ☐ *c)* Las opciones *a)* y *b)* son correctas.
- ☐ *d)* Todas las opciones anteriores son incorrectas.

5. Si se declara un constructor sin parámetros:

- ☐ *a)* Su cuerpo debe estar vacío.
- ☐ *b)* No se podrán crear más constructores.
- ☐ *c)* No se podrá invocar al constructor de la superclase.
- ☐ *d)* El constructor por defecto no se creará.

6. Una clase abstracta:

- ☐ *a)* Obligatoriamente debe tener algún método abstracto.
- ☐ *b)* No puede ser al mismo tiempo *final*.
- ☐ *c)* No puede ser subclase de otra superclase.
- ☐ *d)* No puede tener campos.

7. Una clase abstracta:

- ☐ *a)* Se declara empleando la palabra reservada *classAbstract* antes del nombre de la clase.
- ☐ *b)* No puede tener campos.
- ☐ *c)* Únicamente dispone del constructor por defecto.
- ☐ *d)* Permite generalizar datos de objetos.

8. Los métodos declarados como *default* en una interfaz a partir del JDK 8:

- ☐ *a)* Están disponibles únicamente para instancias de la interfaz.
- ☐ *b)* Están disponibles únicamente para objetos de clases que implementen la interfaz.
- ☐ *c)* Están disponibles para instancias de la interfaz y los objetos de las clases que la implementen.
- ☐ *d)* Todas las opciones anteriores son incorrectas.

9. Una interfaz a partir del JDK 8:

- ☐ *a)* Puede declarar el código de un método estático que estará disponible para las clases que implementen la interfaz.
- ☐ *b)* Puede declarar métodos privados.
- ☐ *c)* Las opciones *a)* y *b)* son correctas.
- ☐ *d)* Todas las opciones anteriores son incorrectas.

10. Una interfaz:

- ☐ *a)* No puede ser instanciada.
- ☐ *b)* Es un tipo de referencia al igual que lo es una clase.
- ☐ *c)* Las opciones *a)* y *b)* son correctas.
- ☐ *d)* Todas las opciones anteriores son incorrectas.

SOLUCIONES:

1.	a	b	**c**	d	5.	a	b	c	**d**						
2.	a	**b**	c	d	6.	a	**b**	c	d	9.	**a**	b	c	d	
3.	a	**b**	c	d	7.	a	b	c	**d**	10.	a	b	**c**	d	
4.	a	b	**c**	d	8.	a	**b**	c	d						

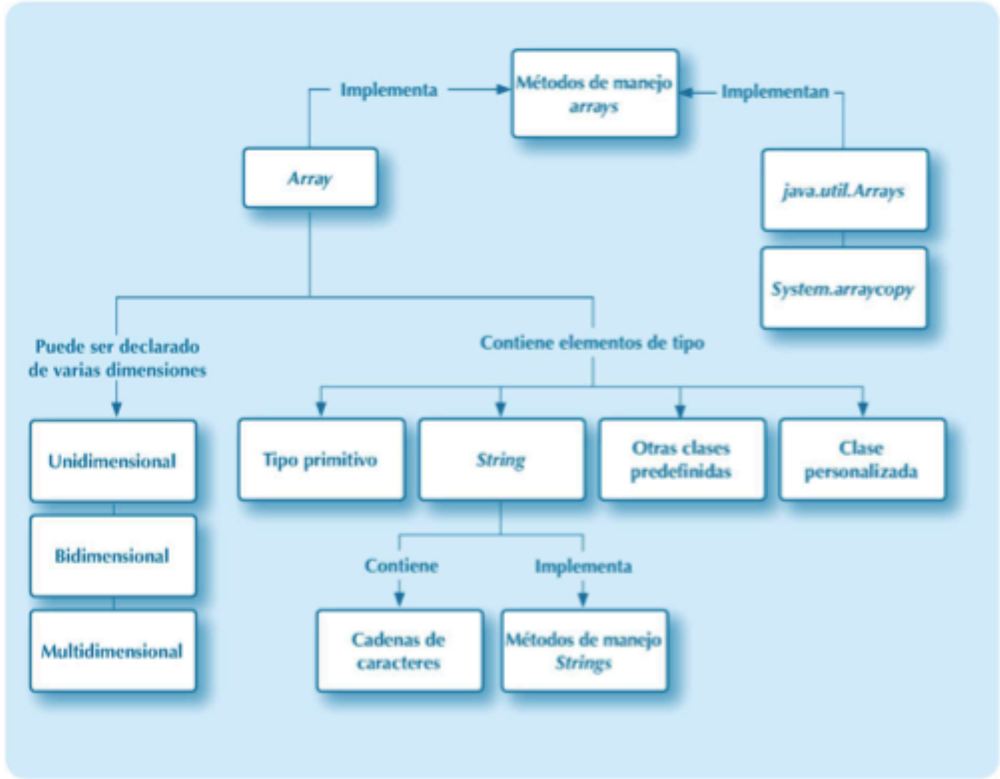
8

Estructuras de almacenamiento: *arrays* y cadenas de caracteres

Objetivos

- ✓ Familiarizarse con las diferentes estructuras de almacenamiento empleadas en Java.
- ✓ Comprender la estructura interna de almacenamiento de un *array* estático.
- ✓ Profundizar en el tipo de datos empleado para almacenar cadenas de caracteres.
- ✓ Conocer los diferentes métodos que permiten manejar los datos de un *array* y de una cadena de caracteres.

Mapa conceptual



Glosario

Array. También conocido como *arreglo* o *matriz* en caso de tener 2 dimensiones, es un conjunto de datos de un tipo determinado, primitivo o no, que se denominan *elementos*. Estos son almacenados juntos y son accedidos a través de un número entero denominado índice.

Cheatsheet. Referencia o guía rápida que permite consultar de forma ágil los contenidos de una temática determinada.

Concatenación. Unión de dos o más elementos.

DNI. Abreviatura de documento nacional de identidad, cuyo identificador se compone de un número formado por 8 dígitos y una letra.

Elemento. Cada uno de los valores contenidos en las posiciones de un *array*. Todos los elementos de un *array* serán del mismo tipo, pudiendo ser este una clase o un tipo primitivo.

Expresión regular. Secuencia de caracteres cuya disposición y orden representa un patrón de búsqueda. Son empleadas a menudo para validar campos en formularios HTML o datos que deben ser procesados y almacenados solo en caso de ser acordes a un determinado formato.

Lexicográfico. Se dice orden lexicográfico como el orden presentado por los caracteres de un determinado alfabeto.

Null. Valor empleado en programación para representar la nada. Cuando un objeto es declarado, pero no inicializado, se dice que apunta a null.

Shallow copy. Copia de un *array* origen hacia un *array* destino en la que únicamente se clona la estructura de almacenamiento y las referencias a los datos en ella incluidas, sin duplicar los espacios de memoria en los que se almacenan los datos en sí. También se conoce como *copia superficial*.

Unicode. Sistema de codificación universal que asocia una secuencia de caracteres a cada carácter a codificar.

8.1. Introducción

Hasta llegar a este capítulo se ha trabajado con tipos de datos primitivos o bien con tipos de datos compuestos de tipos primitivos u otros tipos compuestos a su vez, pero en cualquier caso se ha estado manejando un único dato por cada variable declarada. Estos tipos, que puede parecer suficientes para programas básicos, no disponen de la capacidad necesaria para desarrollos de programas de mayor complejidad. Un ejemplo muy sencillo de entender es el de un programa que tenga que gestionar un número elevado de datos de un mismo tipo para trabajar con ellos de forma recurrente. La alternativa basada en esos tipos primitivos obligaría a declarar tantas variables como datos sea necesario almacenar y a acceder a ellas como variables independientes que serán (el número de variables a declarar será directamente proporcional al número de datos a manejar). Si, por el contrario, es posible emplear un tipo que permita almacenar a todos estos datos juntos, únicamente será necesario declarar una variable.

En Java, al igual que en otros lenguajes de programación modernos, se dispone de tipos especiales que actúan como contenedores de otros elementos. Este tipo se denomina *array* y permite representar a un objeto contenedor de datos de un mismo tipo. Cada ítem del *array* es denominado *elemento*.

8.2. Estructuras

Una estructura es un tipo de dato definido por el programador y que permite agrupar datos relacionados entre sí. Algunos lenguajes como C o Algol, ofrecen formas específicas de definir estos tipos de datos (*struct*). Java, por el contrario, no ofrece esta posibilidad; sin embargo, es posible simular dichas estructuras mediante la declaración de clases para construir tipos compuestos.

Investiga

Busca información sobre los tipos *struct* y *union* que pueden utilizarse en el lenguaje de programación C. ¿Cuál sería la diferencia fundamental entre ambos?

8.3. *Arrays*

Tal como se ha indicado en la introducción, el tipo *array* es un objeto que representa a un contenedor de datos del mismo tipo. Cada ítem del *array* es denominado *elemento*.

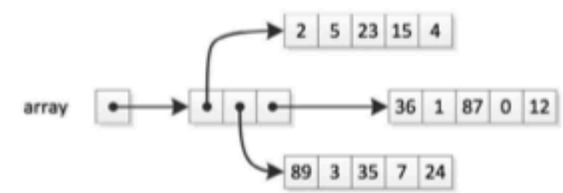
Un *array* puede ser *estático* o *dinámico*. La diferencia radica, conceptualmente hablando, en que en un *array estático* el número de elementos es fijo desde el momento en el que se declara y se reserva el espacio en memoria, mientras que en un *array dinámico* es variable en tiempo de ejecución. En este capítulo únicamente se trabajará con *arrays estáticos*; los *arrays dinámicos* y otras colecciones serán tratados en el capítulo 9 (Colecciones y tipos abstractos de datos).

8.3.1. Dimensiones de un *array*

Un *array* puede almacenar datos en varias dimensiones. En función de ello se distingue entre:

1. Array *unidimensional*, que maneja datos en una única dimensión y por tanto sus elementos pueden representarse de forma lógica, que no física, linealmente:

2. Array *multidimensional*, que se trata de un *array* cuyos elementos son a su vez *arrays*. Si se utilizan dos dimensiones, se puede hablar de *array bidimensional* o *matriz*:



8.3.2. Declaración y creación

Un *array* será declarado indicando *tipo* de los elementos, dimensión y nombre de la variable (los espacios son opcionales). Cada pareja de corchetes ([]) indicará una dimensión. Es posible alternar el orden de la declaración de dimensión y nombre de la variable. En el ejemplo mostrado a continuación se han declarado 5 *arrays* de diferentes dimensiones:

```
short [] edades;     // array unidimensional
int []pesos;         // array unidimensional
int precios [];      // array unidimensional (sintaxis heredada C/C++)
int [][] matriz;     // array bidimensional
byte [][][] arrayTridimensional;     // array tridimensional
```

Una vez declarado es necesario reservar el espacio en memoria de los datos que almacenará:

```
edades = new short[10];
pesos = new int[50];
precios = new int[100];
matriz = new int[4][2];
arrayTridimensional = new byte[4][3][10];
```

8.3.3. Inicialización y acceso

Los *arrays* deberán inicializarse con datos mediante una simple asignación. El acceso a cada elemento de *array* se realizará mediante un índice que marca la posición de este. En un *array* de *N* elementos el primer elemento del *array* será accedido con índice *0*, el segundo con índice *1*… y así sucesivamente hasta el último elemento que será accedido con índice *N-1*. En caso de intentar acceder a elementos que no estén comprendidos en este rango, el sistema lanzará la excepción *java.lang.ArrayIndexOutOfBoundsException*. En el ejemplo siguiente se muestra cómo declarar, inicializar y acceder a los diferentes elementos de un *array*:

```
edades [0]=1;
pesos [1]=3;
precios [1]=2;
matriz[2][1]=200;
arrayTridimensional[2][2][0]=127;
System.out.println(pesos[1]);
System.out.println(matriz[2][1]);
System.out.println(arrayTridimensional[2][2][0]);
```

SABÍAS QUE…

Es posible realizar la declaración, creación e inicialización del *array* al mismo tiempo. Para ello se puede emplear la *sintaxis abreviada:*

```
char[] letras={'a','b','c'};
byte[][] arrayBidimensional = {
    {1, 2, 3},
    {4, 5}
};
```

Gracias a este tipo de declaración se pueden declarar *arrays* multidimensionales cuyos elementos pueden ser *arrays* de distintas dimensiones entre sí.

8.3.4. *Arrays* y métodos

Un *array* puede, por el simple hecho de ser un objeto, ser enviado como argumento a un método o ser devuelto como valor de retorno del método; para ello deberá ser declarado con la misma sintaxis que se empleó para declarar un objeto de tipo *array*:

```
    // invocación a metodo
    int [][] datosRetorno = metodo(edades,pesos,precios,arrayTridimensional);
}
// declaración de metodo
public static int[][] metodo (short [] array1, int []array2,
        int[]array3, byte [][][] array4){
    // ...
    int [][] arrayRetorno = new int[4][2];
    // ...
    return arrayRetorno;
}
```

Actividad propuesta 8.1

Escribe un método que retorne un *array* de 100 números enteros generados aleatoriamente.

8.3.5. Operaciones para realizar sobre un *array* estático

A) *Recorrido del* array

Dado que todo *array* tiene un campo *length* que indica la longitud con la que ha sido creado, es fácil recorrer los elementos mediante un bucle *for*:

```
for (int i=0; i<edades.length;i++) {  // for incremental
    System.out.println(edades[i]);
}
for (int j=edades.length; j>0;j--) {  // for decremental
    System.out.println(edades[j-1]);
}
```

Otra alternativa para recorrer los elementos del *array* es mediante un bucle *for each*:

```
for (int elemento: edades) {  // for each
    System.out.println(elemento);
}
```

Actividad propuesta 8.2

Escribe un método que reciba el *array* generado en la actividad propuesta 8.1 y lo recorra en orden decreciente, calculando el sumatorio de todos los números del *array* y mostrando finalmente dicho valor por pantalla.

B) Búsqueda de elementos en un array

La tecnología Java proporciona la clase *java.util.Arrays* que contiene métodos estáticos y sobrecargados para varios tipos primitivos de datos, con los que poder buscar la posición de un elemento contenido en un *array*:

```
String [] alumnos = ("Paco","Ana","Luis","Rosa","Beatriz");
int posicion= java.util.Arrays.binarySearch(alumnos, "Rosa");
System.out.println("Encontrado en posición: " + posicion);
```

```
run:
Encontrado en posición: 3
```

C) Impresión de elementos de un array

A pesar de que un *array* puede ser recorrido incrementalmente para conocer los elementos en él contenidos, la clase *java.util.Arrays* proporciona el método *toString*, que permite obtener la secuencia de caracteres que representa al *array* en un formato legible. Como argumento es necesario pasar un objeto de tipo *array*:

```
char A []  = ('a','b','c');
System.out.println(Arrays.toString(A));
```

```
run:
[a, b, c]
```

D) Ordenación de elementos de un array

La clase *java.util.Arrays* también permite ordenar ascendentemente los elementos del *array*:

```
int [] numeros = (121,12,33,1,55);
java.util.Arrays.sort(numeros);
System.out.println("Array ordenado: " + Arrays.toString(numeros));
```

```
run:
Array ordenado: [1, 12, 33, 55, 121]
```

E) Comparación arrays

Además de los métodos anteriormente descritos de la clase *java.util.Arrays*, existe también el método *equals* que permiten comparar el contenido de dos *arrays*:


```
int array1[] = (4,7,14,9);
int array2[] = array1;
int arrayClonado[] = array1.clone();
System.out.println(array1 == arrayClonado);
System.out.println(array1 == array2);
System.out.println(java.util.Arrays.equals(array1, arrayClonado));
```

```
run:
false
true
true
```

NOTA: Es importante destacar que la comparación realizada con el operador binario de igualdad "==" compara si los objetos de tipo *array* apuntan a la misma dirección de memoria, mientras que el método *equals* compara si ambos *arrays* tienen los mismos elementos.

F) Inicialización masiva de elementos de un array

El último de los métodos mostrados de la clase *java.util.Arrays* es el médodo *fill* que permite rellenar un *array* con los valores pasados como argumentos:

```
int [][] array = new int[3][6];
java.util.Arrays.fill(array[1],1);
java.util.Arrays.fill(array[2],2);
System.out.println("Array[0]: " + Arrays.toString(array[0]));
System.out.println("Array[1]: " + Arrays.toString(array[1]));
System.out.println("Array[2]: " + Arrays.toString(array[2]));
```

```
run:
Array[0]: [0, 0, 0, 0, 0, 0]
Array[1]: [1, 1, 1, 1, 1, 1]
Array[2]: [2, 2, 2, 2, 2, 2]
```

G) Copia de arrays

Existen varias formas de realizar copias de un *array*. En este apartado se exponen y describen las principales, haciendo especial énfasis en los diferentes resultados en función del tipo de datos de los elementos contenidos.

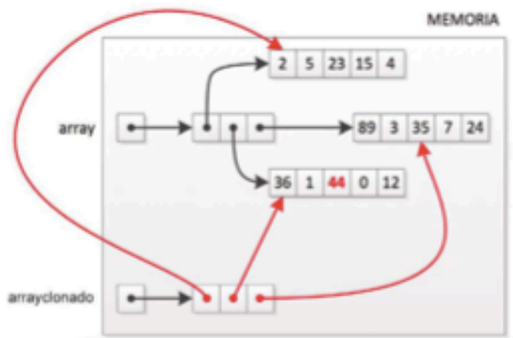
1. Método *clone*

El método *clone* devuelve una copia exacta del *array* desde el cual se realiza la invocación. Hay que tener en cuenta que el resultado de la invocación será distinto dependiendo del tipo de los datos contenidos en el *array*. Si se trata de *tipos primitivos*, la invocación de *clone* devolverá un nuevo *array* cuyos datos serán totalmente independientes de los originales (los cambios en los datos del *array original* no afectarán al *array clonado*). En el siguiente ejemplo puede comprobarse cómo se lleva a cabo la copia de un *array* de números enteros:

```
int array[] = {4,7,14,9};
int arrayClonado[] = array.clone();
System.out.println(array == arrayClonado);
arrayClonado[0] = 0;
System.out.println(array [0]);
System.out.println(arrayClonado [0]);
```

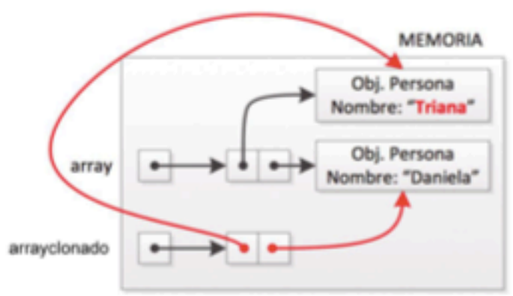
```
run:
false
4
0
```

Sin embargo, si los *tipos son no primitivos*, no se obtendrá una copia exacta sino una *Shallow copy*. Esto es debido a que los datos contenidos en el *array* original son referencias a los objetos que contienen los datos reales. Al duplicarse, lo que se está clonando es el *array* de referencias, pero estas seguirán apuntando a los mismos datos. A continuación, se muestran dos ejemplos, uno con un *array bidimensional* y otro con un *array* contenedor de objetos de la clase *Persona*:




```
int array[][] = {{2,5,23,15,4}, {36,1,87,0,12}, {89,3,35,7,24}};
int arrayClonado[][] = array.clone();
System.out.println(array == arrayClonado);
System.out.println(array [1][2]);
arrayClonado[1][2] = 44;
System.out.println(array [1][2]);
System.out.println(arrayClonado [1][2]);
```

```
run:
false
87
44
44
```




```
Persona array[] = {new Persona("Juan"), new Persona("Daniela")};
Persona arrayClonado[] = array.clone();
System.out.println(array == arrayClonado);
System.out.println(array [0].nombre);
arrayClonado[0].nombre = "Triana";
System.out.println(array [0].nombre);
System.out.println(arrayClonado [0].nombre);
```

```
run:
false
Juan
Triana
Triana
```

2. Métodos *Arrays.copyOf* y *Arrays.copyOfRange*

Los métodos *copyOf* y *copyOfRange* de la clase *java.util.Arrays* retornarán una copia total o parcial del *array* pasado como argumento (no será necesario reservar el espacio en el *array* destino antes). La sintaxis de uso es la siguiente:

```
type[] Arrays.copyOfRange (arrayOrigen, posInicio, posFinal);
type[] Arrays.copyOf (arrayOrigen, length);
```

donde:

- ✓ *type* es el tipo del dato del *array*.
- ✓ *arrayOrigen* será el *array* desde el que se quiere copiar los datos.
- ✓ *posInicio* el primer elemento desde el que comenzar la copia.
- ✓ *posFinal* el último elemento a copiar.
- ✓ *length* el número de elementos a copiar desde la primera posición.

A continuación, se puede ver un ejemplo de uso de ambos métodos:

```
int[] a = {1,2,3,4,5};
int[] b = Arrays.copyOfRange(a, 0, a.length); // copia completa de a en b
int[] c = Arrays.copyOfRange(a, a.length-2, a.length); // copia los 2 últimos elementos
int[] d = Arrays.copyOf(a, a.length); // copia completa de a en d
int[] e = Arrays.copyOf(a, 2); // copia los 2 primeros elementos
System.out.println("Array a: " + Arrays.toString(a));
System.out.println("Array b: " + Arrays.toString(b));
System.out.println("Array c: " + Arrays.toString(c));
System.out.println("Array d: " + Arrays.toString(d));
System.out.println("Array e: " + Arrays.toString(e));
```

```
run:
Array a: [1, 2, 3, 4, 5]
Array b: [1, 2, 3, 4, 5]
Array c: [4, 5]
Array d: [1, 2, 3, 4, 5]
Array e: [1, 2]
```

Al igual que ocurría al emplear el método *clone*, si los tipos no son primitivos se obtendrá una *Shallow copy*.

3. Método *System.arraycopy*

El método *arraycopy* también permite copiar los datos de un *array* a otro de forma total o parcial. A diferencia de los métodos anteriores, para utilizar este método es necesario reservar el espacio en el *array* destino antes de realizar la copia. La sintaxis de uso es la siguiente:

```
arraycopy (Object src, int srcPos, Object dest, int destPos, int length)
```

en donde:

- ✓ *src* hace referencia al *array* origen.

- ✓ *dest* hace referencia al *array* destino.
- ✓ *srcPos* hace referencia al elemento del *array* origen desde el que se comienza la copia.
- ✓ *destPos* hace referencia a la posición en la que se comenzará copia en el destino.
- ✓ *length* hace referencia al número de elementos a copiar desde el *array* origen.

A continuación, se muestra un ejemplo de uso del método *arraycopy*:

```
int[] array1 = (1,2,3,4,5);
int[] array2 = new int[5];
System.arraycopy(array1, 2, array2, 1, 2);
for (int i=0;i<array2.length;i++)
    System.out.println(array2 [i]);
```

```
run:
0
3
4
0
0
```

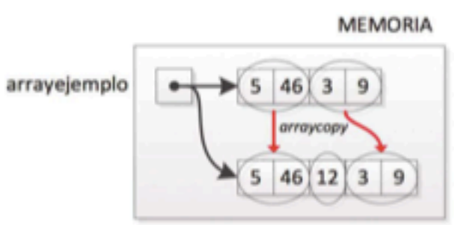
Al igual que ocurría al emplear el método *clone*, si los tipos no son primitivos, se obtendrá una *Shallow copy*.

PARA SABER MÁS

Además de los métodos expuestos en este capítulo existen otros mecanismos para llevar a cabo la copia de un *array* como la API *Stream* o la *Serialización*.

H) Insertar elementos en un array

Como ya se explicó anteriormente, uno de los inconvenientes que presentan los *arrays* estáticos es la rigidez que ofrecen para modificar su tamaño. Es por ello por lo que, si se quiere insertar un nuevo elemento en un *array* estático, será necesario declarar un nuevo *array* de dimensiones mayores al original y sobre este realizar la inserción y copiar los elementos ya existentes, sobrescribiendo finalmente la referencia al *array* original por la referencia al nuevo. Gracias a las utilidades expuestas en el apartado anterior, esta tarea es sumamente sencilla tal como puede comprobarse en el siguiente ejemplo:




```
public static int[] insertarElemento (int[] array, int elemento, int posicion){
    int [] nuevoArray = new int [array.length+1];
    System.arraycopy(array, 0, nuevoArray, 0, posicion);
    nuevoArray[posicion] = elemento;
    System.arraycopy(array, posicion, nuevoArray, posicion+1, array.length-posicion);
    return nuevoArray;
}
public static void main(String[] args) {
    int[] arrayEjemplo = (5, 46, 3, 9);
    int posicion = 2;
    int nuevoElemento = 12;
    System.out.println("Array original: "+Arrays.toString(arrayEjemplo));
    arrayEjemplo = insertarElemento (arrayEjemplo, nuevoElemento, posicion);
    System.out.println("Array con nuevo elemento: "+Arrays.toString(arrayEjemplo));
}
```

```
run:
Array original: [5, 46, 3, 9]
Array con nuevo elemento: [5, 46, 12, 3, 9]
```

I) Eliminar elementos de un array

La tarea opuesta a la inserción se puede llevar a cabo de forma similar mediante el empleo de las utilidades ya mencionadas. A continuación se muestra un ejemplo de implementación de un programa que contiene un método para eliminar elementos del *array*:

```
public static int[] eliminarElemento (int[] array, int posicion){
    int [] nuevoArray = new int [array.length-1];
    System.arraycopy(array, 0, nuevoArray, 0, posicion);
    System.arraycopy(array, posicion+1, nuevoArray, posicion, nuevoArray.length-posicion);
    return nuevoArray;
}
public static void main(String[] args) {
    int[] arrayEjemplo = (5, 46, 3, 9);
    int posicion = 1;
    System.out.println("Array original: "+Arrays.toString(arrayEjemplo));
    arrayEjemplo = eliminarElemento (arrayEjemplo, posicion);
    System.out.println("Array con nuevo elemento: "+Arrays.toString(arrayEjemplo));
}
```

```
run:
Array original: [5, 46, 3, 9]
Array con nuevo elemento: [5, 3, 9]
```

Actividad propuesta 8.3

Desarrolla un programa que pida por consola el número de alumnos de una clase y que a continuación solicite los *N* nombres para almacenarlos en un *array*. A continuación, implementa los métodos necesarios para eliminar a un alumno del *array* a partir de su nombre, para añadir un alumno nuevo al *array* de alumnos y para ordenar el listado de alumnos.

8.4. Cadenas de caracteres

Las cadenas de caracteres ya fueron introducidas en el capítulo 4. Sin embargo, no se ha profundizado en los métodos de los que este tipo de objetos dispone. Será en este capítulo donde se hará una descripción detallada de cómo utilizar estos objetos.

8.4.1. Declaración, creación e inicialización

Después de ver en detalle qué es un *array*, posiblemente se piense en la posibilidad de manejar cadenas de caracteres como *arrays* formados de elementos tipo *char*. Sin embargo, el lenguaje Java ofrece el tipo *String*, que ha sido especialmente diseñado para manejar cadenas.

Existen varias formas de construir un *String*. A continuación se muestra un ejemplo en el que se construye a partir de una secuencia de caracteres encerrados entre comillas dobles ("") y a través de un *array* de elementos tipo *char*:

```
String cadena1 = "Hola mundo!";
System.out.println(cadena1);

char[] arrayChar = { 'H','o','l','a',' ','m','u','n','d','o','!'};
String cadena2 = new String(arrayChar);
System.out.println(cadena2);
```

```
run:
Hola mundo!
Hola mundo!
```

Sabías que...

Una vez que se crea e inicializa un *String*, este es inmutable y no se puede modificar.

8.4.2. Operaciones

Existe un gran repertorio de métodos públicos que permiten llevar a cabo operaciones sobre objetos de tipo *String*. En el cuadro 8.1 se halla un resumen de los más destacados (algunos de ellos están sobrecargados):

Cuadro 8.1
Métodos de *String* más destacados

`char charAt(int posición)`	Retorna el carácter situado en posición.
`int compareTo(String cadena)`	Compara lexicográficamente con la cadena que se pasa como argumento basándose en los valores Unicode de cada carácter. Un *valor negativo* indica que el *String* precede lexicográficamente a la cadena. Un *valor positivo* indicará lo contrario. Un *valor 0* indica que ambos *Strings* son iguales.

[.../...]

Cuadro 8.1 (cont.)

`int compareToIgnoreCase(String cadena)`	Similar al anterior pero sin tener en cuenta las posibles diferencias entre mayúsculas y minúsculas.
`String concat(String cadena)`	Retorna la concatenación del *String* invocante con la cadena que se pasa como argumento. El resultado es equivalente a emplear el operador '+' con ambas cadenas.
`boolean contains(String s)`	Devuelve *true* si el *String* contiene la cadena pasada como argumento, y *false* en caso contrario.
`boolean endsWith(String sufijo)`	Devuelve *true* si el *String* finaliza con la cadena pasada como argumento, y *false* en caso contrario.
`boolean equals(Object objeto)`	Devuelve *true* si el *String* pasado como argumento es igual, lexicográficamente hablando, al que realiza la invocación.
`boolean equalsIgnoreCase(String cadena)`	Similar al anterior pero sin tener en cuenta mayúsculas o minúsculas.
`int indexOf(String cadena)`	Devuelve un *número entero* que representa la posición de la primera ocurrencia de cadena dentro del *String*. Si no se encuentra devolverá *-1*.
`int indexOf(String cadena, int posición)`	Similar al anterior comenzando la búsqueda en *posición*.
`int lastIndexOf(String cadena)`	Similar pero buscando en orden inverso desde el final.
`int lastIndexOf(String str, int posición)`	Similar al anterior comenzando la búsqueda desde *posición* hacia atrás.
`boolean isEmpty()`	Devuelve *false* en caso de que el *String* no contenga caracteres. Si el objeto es *null*, retornará una excepción.
`int length()`	Devuelve el número de caracteres del *String*.
`boolean matches(String regex)`	Devuelve *true* si el *String* coincide con la expresión regular pasada como argumento.
`String replace(String cadenaBuscada, String reemplazo)`	Sustituye cada ocurrencia de cadena buscada por la cadena de reemplazo.
`String replaceAll(String regex, String reemplazo)`	Similar al anterior pero buscando secuencias que coinciden con una *expresión regular*.
`String replaceFirst(String regex, String reemplazo)`	Similar al anterior pero solo sustituye la primera ocurrencia.
`String[] split(String regex)`	Busca las coincidencias de la *expresión regular* en la cadena y retorna un *array* de las subcadenas comprendidas entre ellas.
`boolean startsWith (String prefijo)`	Devuelve *true* si el *String* comienza con la cadena pasada como argumento.
`String substring(int posComienzo)`	Retorna la subcadena comprendida desde *posComienzo* hasta el final de la cadena.

[.../...]

CUADRO 8.1 (CONT.)

`String substring(int comienzo, int final)`	Retorna la subcadena comprendida entre las posiciones comienzo y final.
`char[] toCharArray()`	Devuelve un *array* cuyos elementos son cada uno de los caracteres contenidos en el *String*.
`String toLowerCase()`	Retorna el mismo *String* con todos los caracteres en minúsculas.
`String toUpperCase()`	Similar al anterior pero convirtiéndolos en mayúsculas.
`String trim()`	Retorna una copia de la cadena en la que se han eliminado los posibles espacios en blanco del comienzo y del final.

En el siguiente ejemplo puede verse el manejo de algunos de los métodos descritos:

```
String cadena = "  Esta es una cadena    ";
System.out.println(cadena.trim());
System.out.println(cadena.toLowerCase());
System.out.println("Caracter en posición 3: "+cadena.charAt(3));

String copiaCadena = cadena;
System.out.println("cadena --> "+cadena);
System.out.println("copiaCadena --> "+copiaCadena);
System.out.println("cadena == copiaCadena --> "+(cadena==copiaCadena));
System.out.println("cadena.equals(copiaCadena) --> "+(cadena.equals(copiaCadena)));

cadena = cadena.replace("cadena", "cadena de caracteres");

System.out.println("cadena --> "+cadena);
System.out.println("copiaCadena --> "+copiaCadena);
System.out.println("cadena == copiaCadena --> "+(cadena==copiaCadena));
System.out.println("cadena.equals(copiaCadena) --> "+(cadena.equals(copiaCadena)));
```

```
run:
Esta es una cadena
  esta es una cadena
Carácter en posición 3: s
cadena -->   Esta es una cadena
copiaCadena -->   Esta es una cadena
cadena == copiaCadena --> true
cadena.equals(copiaCadena) --> true
cadena -->   Esta es una cadena de caracteres
copiaCadena -->   Esta es una cadena
cadena == copiaCadena --> false
cadena.equals(copiaCadena) --> false
```

Actividad propuesta 8.4

Escribe un programa que sea capaz de contar el número de palabras diferentes que hay en un texto que se le pasa por argumento, sin tener en cuenta si están escritas en mayúsculas o minúsculas.

Recursos web

Las expresiones regulares son ampliamente utilizadas en programación para conseguir validar datos y realizar búsquedas de patrones en textos. En la URL https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html podrás obtener una detallada descripción de cómo pueden ser utilizadas en un programa Java. Además, hay disponibles varias herramientas web que ayudan a construir y validar las expresiones regulares (https://www.regexplanet.com/advanced/java/index.html y http://myregexp.com/ son algunas de ellas).

8.4.3. Conversiones

Por medio de las clases envoltorio que fueron expuestas en el capítulo 4, es posible realizar conversiones de dígitos a cadenas de caracteres y viceversa. En el cuadro 8.2 se muestra un resumen de los diferentes mecanismos disponibles para hacer las conversiones.

CUADRO 8.2
Conversiones de dígitos a *String* y viceversa

De *String* al tipo numérico primitivo	De *String* al tipo numérico representado por la clase envoltorio
`Byte.parseByte(cadena);`	`Byte.valueOf(cadena);`
`Short.parseShort(cadena);`	`Short.valueOf(cadena);`
`Integer.parseInt(cadena);`	`Integer.valueOf(cadena);`
`Long.parseLong(cadena);`	`Long.valueOf(cadena);`
`Float.parseFloat(cadena);`	`Float.valueOf(cadena);`
`Double.parseDouble(cadena);`	`Double.valueOf(cadena);`
De tipo numérico a *String*	
variableNumerica+""	*String.valueOf(variableNumerica);*
Método toString() de la clase envoltorio.	

Ejemplo de uso de conversores de tipos numéricos a *String*:

```
int num1 = 1;
Integer num2 = new Integer (2);
System.out.println(String.valueOf(num1));
System.out.println(String.valueOf(num2));
System.out.println(num1+"");
System.out.println(num2+"");
System.out.println(num2.toString());
```

```
run:
1
2
1
2
2
```

Resumen

- Los *arrays* estáticos permiten almacenar elementos de tipos primitivos o clases.
- A diferencia de otros lenguajes, Java no dispone de estructuras y simula este tipo de elementos mediante el uso de clases.
- Existen varios mecanismos para realizar copias de un *array* y cuando un *array* contiene objetos, y no datos de tipo primitivo, las copias son en realidad copias superficiales.
- Un *array* estático no puede modificar su tamaño en tiempo de ejecución.
- Las cadenas de caracteres se representan mediante objetos de la clase *String*, la cual ofrece varios métodos para trabajar con las cadenas.

Ejercicios propuestos

1. Diseña tu propio *Cheatsheet* con los métodos explicados a lo largo del capítulo.
2. Describe la diferencia existente entre utilizar el operador "+" y el método *concat* sobre objetos de tipo *String*.
3. ¿Qué método emplearías para sustituir todas las ocurrencias de correos electrónicos dentro de un texto?
4. Una vez se crea e inicializa un *String*, por ejemplo *String a="cadena";*, este es inmutable y no se puede modificar a lo largo de toda su vida. ¿Qué sucede al realizar una operación de asignación a otra cadena como la siguiente *a="nueva cadena";*? Razona tu respuesta.
5. Propón ejemplos reales en los que puede ser interesante hacer uso de los métodos *replace* y *split* de la clase *String*.
6. Investiga sobre el uso del método *trim* de la clase *String*. Además de eliminar espacios del comienzo y el final de una cadena, ¿eres capaz de eliminar tabulaciones?

Supuestos prácticos

1. Construye un programa que genere 100 números aleatorios mediante el uso de la función *Math.random* y que posteriormente ofrezca al usuario la posibilidad de:
   - Conocer el mayor de los números.
   - Conocer el menor de los números.
   - Obtener la suma de todos los números.
   - Obtener la media de los números.
   - Sustituir el valor de un elemento por otro número introducido por teclado.
2. Implementa un programa que pregunte por un DNI y valida si se trata de un DNI válido. El cálculo de la letra correspondiente a los dígitos se realiza mediante algoritmo descrito en http://www.interior.gob.es/web/servicios-al-ciudadano/dni/calculo-del-digito-de-control-del-nif-nie.
3. Escribe el código de un programa que pida por teclado el contenido de un texto y realiza búsquedas de una determinada palabra dentro de él. La secuencia de ejecución será: pedir texto, pedir palabra, buscar número de ocurrencias de palabra en texto, mostrar por pantalla el número de apariciones de la palabra.
4. Supón que tu empresa se va a hacer cargo del desarrollo de un programa para el registro de datos de los trabajadores de una compañía. Esta tiene varias sedes que están identificadas por el nombre de la ciudad en la que residen (hasta el momento *Madrid, Barcelona, Valencia* y *Oviedo*). En cada sede hay varios departamentos, algunos de los cuales están repetidos entre varias sedes, a los que pertenecen los diferentes empleados. Hasta el momento los departamentos existentes en todas las sedes son *Ventas, RR. HH.* y *Producción*. De cada empleado es importante almacenar el *código de empleado, DNI, nombre, primer apellido, segundo apellido, año de nacimiento (numérico)* y si tiene concedida, o no, *reducción de jornada (booleano)*. Diseña la estructura que almacenará los datos de la compañía y escribe el código fuente del programa que permita comenzar a registrar la información de los empleados.

Actividades de autoevaluación

1. Si se ejecuta *arrayB = arrayA.clone();* y los datos contenidos son elementos de tipo char:
   - ☐ *a)* La instrucción *arrayA[1] = 'z';* será equivalente a *arrayB[1] = 'z';*
   - ☐ *b)* La instrucción *arrayA[1] = 'z';* no altera el contenido de *arrayB*
   - ☐ *c)* La instrucción *arrayA[1] = 'z';* creará un nuevo objeto tipo *Character* para ser almacenado en arrayA.
   - ☐ *d)* Todas las opciones son incorrectas.
2. Si se ejecuta *arrayB = arrayA;* y los datos contenidos son elementos de tipo *char*:
   - ☐ *a)* La instrucción *arrayA[1] = 'z';* será equivalente a *arrayB[1] = 'z';*
   - ☐ *b)* La instrucción *arrayA[1] = 'z';* no altera el contenido de *arrayB*
   - ☐ *c)* *arrayA* y *arrayB* son totalmente independientes.
   - ☐ *d)* Todas las opciones son incorrectas.

3. ¿Cuál de los siguientes métodos requiere de reserva de espacio en memoria previa para hacer la copia?
   - ☐ *a)* *Arrays.copyOf*
   - ☐ *b)* *Arrays.copyOfRange*
   - ☐ *c)* *System.arraycopy*
   - ☐ *d)* *clone*

4. Para "trocear" una cadena de texto en varias partes que vienen separadas por el carácter "#"dentro de la cadena original, se podrá emplear:
   - ☐ *a)* El método *split* de la clase *String*.
   - ☐ *b)* Una combinación de los métodos *indexOf* y *substring* de la clase *String*.
   - ☐ *c)* Una combinación de los métodos *lastIndexOf* y *substring* de la clase *String*.
   - ☐ *d)* Todas las opciones son correctas.

5. Identifica la afirmación correcta entre las siguientes:
   - ☐ *a)* El método *length* de un *array* devuelve el número de elementos en él contenidos.
   - ☐ *b)* El campo *length* de un *String* devuelve el número de caracteres en él contenidos.
   - ☐ *c)* A partir de un *array* se puede invocar a su método *toString* para obtener una cadena de texto que contenga los elementos en el array contenidos.
   - ☐ *d)* Todas las opciones son incorrectas.

6. ¿Cuál de las siguientes instrucciones no realizará una conversión a *String*?
   - ☐ *a)* *String.valueOf(num1);*
   - ☐ *b)* *String.parseString(num1);*
   - ☐ *c)* *num1+""+num2.*
   - ☐ *d)* *num1.toString()+num2.toString();*

7. ¿Cuál de los siguientes métodos permite obtener un número entero, de tipo primitivo, a partir de una cadena de caracteres?
   - ☐ *a)* *Integer.valueOf (cadena);*
   - ☐ *b)* *Integer.value (cadena);*
   - ☐ *c)* *Integer.parseInt (cadena);*
   - ☐ *d)* *Integer.parse(cadena);*

8. ¿Qué método busca coincidencias en una cadena de una expresión regular pasada como argumento y retorna un *array* de las subcadenas comprendidas entre ellas?
   - ☐ *a)* *matches*
   - ☐ *b)* *substring*
   - ☐ *c)* *split*
   - ☐ *d)* *trim*

9. ¿Cuál de las siguientes declaraciones se realiza conforme a la sintaxis del lenguaje C/C++?
   - ☐ *a)* *char cadena [];*
   - ☐ *b)* *char [] cadena;*
   - ☐ *c)* *char []cadena;*
   - ☐ *d)* *[]char cadena;*

10. ¿Qué valor tomará la variable s tras la ejecución *s = 3+2+"4"+3+2;*?
    - ☐ *a)* 545.
    - ☐ *b)* 5432.
    - ☐ *c)* 32432.
    - ☐ *d)* 14.

SOLUCIONES:

1.	a	**b**	c	d	**5.**	a	b	c	**d**						
2.	**a**	b	c	d	**6.**	a	**b**	c	d	**9.**	**a**	b	c	d	
3.	a	b	**c**	d	**7.**	a	b	**c**	d	**10.**	a	**b**	c	d	
4.	a	b	c	**d**	**8.**	a	b	**c**	d						

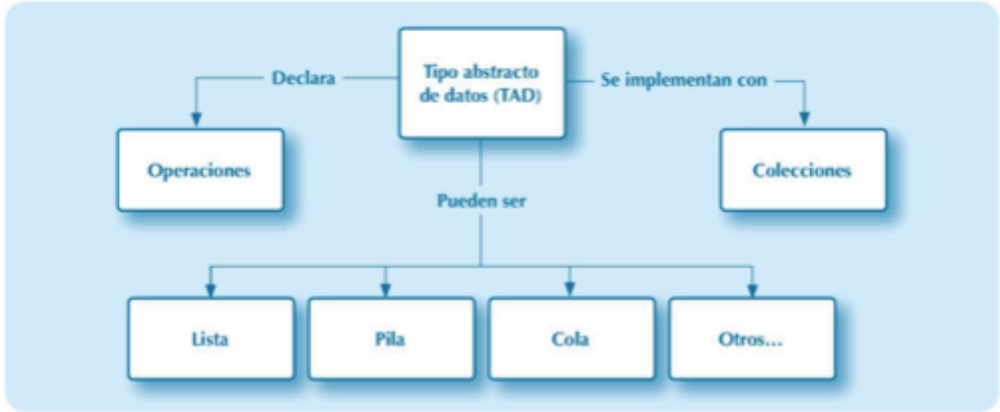
9

Colecciones y tipos abstractos de datos

Objetivos

- ✓ Conocer las diferentes alternativas proporcionadas en Java para el uso de colecciones.
- ✓ Aprender a manejar los tipos de datos colección.
- ✓ Dominar los diferentes tipos abstractos de datos para almacenar información y las operaciones típicas que se realizan sobre estos.
- ✓ Asociar cada colección a cada tipo abstracto de datos y saber aplicar la más adecuada en cada situación.

Mapa conceptual



Glosario

Árbol rojo-negro. Estructura de almacenamiento basada en árboles binarios cuyos datos son comparables para poder establecer un orden.

Cola. Estructura de datos en la que el primer elemento en entrar es el primero en salir (denominadas estructuras *FIFO*).

Colección. Objeto que representa un grupo de elementos de tamaño indeterminado y que implementa varios métodos para el manejo de dichos elementos. Permite implementar un tipo abstracto de datos.

Customizar. Personalizar, modificar o adaptar a las necesidades o preferencias de quien lo utiliza.

First In First Out (FIFO). Es la forma en la que se manejan los datos almacenados en un TAD cola y que se caracteriza por ofrecer acceso siempre al elemento que más tiempo lleva almacenado.

Iterador. Elemento empleado en programación para recorrer todos los elementos existentes en la estructura de almacenamiento de un tipo abstracto de dato.

Last In First Out (LIFO). Es la forma en la que se manejan los datos almacenados en un TAD pila y que se caracteriza por ofrecer acceso siempre al elemento que menos tiempo lleva almacenado.

Lista. Estructura de datos que permite el acceso a cualquier posición de dicha estructura para la consulta, modificación o borrado de elementos.

Mapa. También conocido como *array* asociativo o diccionario, es una estructura de almacenamiento cuyos valores son recuperados a través de secuencias denominadas clave.

Pila. Estructura de datos en la que el último elemento en entrar es el primero en salir (denominadas estructuras LIFO).

Tabla *hash*. Estructura de almacenamiento que asocia elementos clave con elementos valor. Las claves son convertidas en un índice por medio de una función *hash* o resumen que retorna un valor numérico correspondiente a la dirección en la que se encuentra almacenado el valor.

Tipo abstracto de dato (TAD). Conjunto de valores y operaciones que permiten trabajar sobre ellos manteniendo la estructura de almacenamiento definida (lista, pila, cola…).

***Top Of Stack* (TOS).** Elemento cima de una pila, el único sobre el que se tiene acceso tanto para recuperarlo como para apilar nuevos elementos encima de él.

9.1. Colecciones y tipos abstractos de datos

En el capítulo anterior se ha trabajado con diferentes elementos que permiten guardar la información utilizada por el programa. Algunos de ellos permiten almacenar valores simples, entendiéndose simple como un único dato, mientras que otros posibilitan almacenar varios valores. Sin embargo, algunas de estas estructuras presentan restricciones a la hora de ampliar o disminuir el número de elementos y tienen que ser paliadas mediante la redeclaración de nuevas estructuras de distinto tamaño y el trasvase de datos entre ambas. Para evitar tener que realizar estas operaciones de forma manual, Java dispone de otras estructuras dinámicas que facilitan el desarrollo y que son conocidas como colecciones.

Definiremos una *colección* como un objeto que representa un grupo sin tamaño determinado de elementos. Los objetos que se comportan como colecciones implementan métodos que han sido definidos en interfaces bien conocidas. Entre las ventajas inherentes al empleo de colecciones se encuentra la disminución del esfuerzo necesario para desarrollar nuevas aplicaciones y la facilidad aportada para la construcción de *tipos abstractos de datos* (*lista*, *pila*, *cola*…).

En la figura 9.1 puede verse la relación existente entre las diferentes interfaces (en cursiva) y las clases (en negrita) que las implementan. Algunas clases abstractas que se encuentran entre ellas han sido obviadas para facilitar la comprensión.

Actividad propuesta 9.1

¿Qué diferencias y similitudes existen entre un *array* y una *colección*?

Recuerda

✓ Para poder emplear las clases que implementan las colecciones es necesario importar el paquete que las contiene al comienzo del programa. Todas ellas se encuentran en *java.util*.

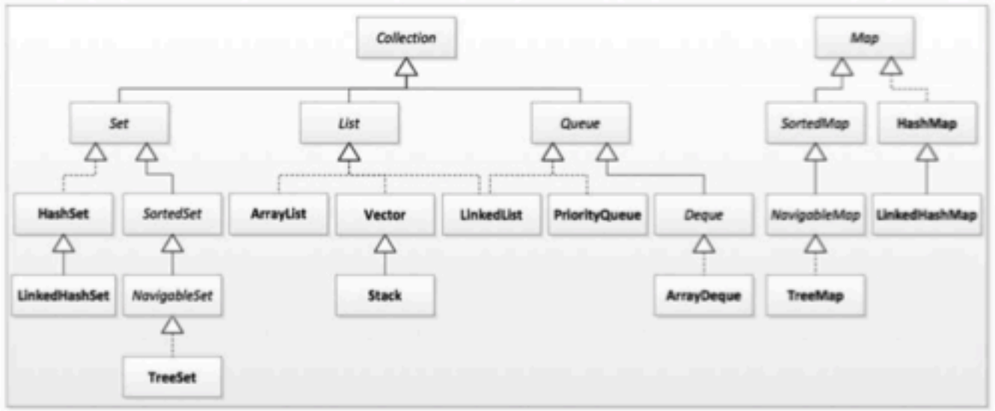


Figura 9.1
Colecciones Java.

9.2. Interfaz *Collection*

La interfaz *Collection* es la interfaz raíz de las colecciones basadas en *Set*, *List* o *Queue*. Hereda de la interfaz *Iterable*, motivo por el cual los elementos contenidos podrán ser recorridos utilizando un iterador. Cada una de las colecciones que se implementan a partir de esta interfaz tiene unas características únicas en cuanto a la posibilidad de almacenar objetos duplicados, orden de los objetos… A continuación, se enumeran y describen los métodos que todas las colecciones que implementan *Collection* tienen disponible:

- `int size()` Devuelve el número de elementos contenidos en la colección.
- `boolean isEmpty()` Devuelve *true* si no existen elementos en la colección.
- `void clear()` Vacía la colección.
- `boolean contains(Object elemento)` Retorna *true* en caso de que el elemento ya exista en la colección.
- `boolean containsAll(Collection<?> c2)` Retorna *true* en caso de que *c2* sea un subconjunto de la colección.
- `boolean add(E elemento)` Añade el elemento que se le pasa por argumento y devuelve *true* si la colección se ha modificado.
- `boolean addAll(Collection<? extends E> c2)` Añade a la colección todos los elementos de la colección *c2*. Retorna *true* si la colección original ha sido modificada.
- `boolean remove(Object elemento)` Elimina el elemento que se le pasa por argumento. Retorna *true* si el elemento ha podido eliminarse o *false* en caso contrario.
- `boolean removeAll(Collection<?> c2)` Elimina de la colección los elementos presentes en la colección *c2*, devolviendo *true* si la colección se ve modificada.
- `boolean retainAll(Collection<?> c2)` Elimina de la colección los elementos que no están presentes en la colección *c2*, devolviendo *true* si la colección se ve modificada.
- `Iterator<E> iterator()` Retorna un iterador sobre la colección de elementos tipo *E* para recorrerla en orden.
- `Object[] toArray()` Devuelve un *array* que contiene los elementos de la colección.

9.2.1. Interfaz *Set*

Se trata de la interfaz que se utilizará para implementar conjuntos de elementos. Existen tres colecciones que implementan la interfaz *Set*. En todas ellas se dispone de los métodos declarados en *Collection* añadiéndose las restricciones necesarias para que los objetos no puedan ser duplicados.

A) HashSet

Esta implementación de la colección *Set* almacena los elementos en una tabla *hash* (concretamente con *HashMap*). Es la implementación que permite el acceso más rápido a los elementos. Por el contrario, no permite realizar accesos secuenciales y los elementos no están ordenados por ningún criterio. Ejemplo de uso:

```
// HashSet
HashSet<String> setNombres=new HashSet<>();
setNombres.add("Juan");
setNombres.add("Micaela");
setNombres.add(null);
setNombres.add("Micaela");
setNombres.add("David");
// Recorriendo elementos
Iterator<String> itr=setNombres.iterator();
while(itr.hasNext()){
    System.out.println(itr.next());
}
// Imprimiendo todo el conjunto
System.out.println(setNombres);
```

```
run:
null
David
Juan
Micaela
[null, David, Juan, Micaela]
```

B) LinkedHashSet

Es una tabla *hash* con enlaces que permite almacenar los elementos ordenados según han sido insertados. Si se sustituye la colección del ejemplo anterior por *LinkedHashSet*, el resultado es:

```
// LinkedHashSet
LinkedHashSet<String> setNombres=new LinkedHashSet<>();
setNombres.add("Juan");
setNombres.add("Micaela");
setNombres.add(null);
setNombres.add("Micaela");
setNombres.add("David");
// Recorriendo elementos
Iterator<String> itr=setNombres.iterator();
while(itr.hasNext()){
    System.out.println(itr.next());
}
// Imprimiendo todo el conjunto
System.out.println(setNombres);
```

```
run:
Juan
Micaela
null
David
[Juan, Micaela, null, David]
```


C) TreeSet

La implementación *TreeSet* almacena los elementos en orden ascendente según sus valores utilizando *árboles binarios rojo-negro*. A diferencia de las anteriores, no puede contener *null*. Es más lenta que *HashSet*, pero ofrece algunas funcionalidades nuevas:

- ✓ `E first()` Devuelve el menor elemento que hay en la colección.
- ✓ `E last()` Devuelve el mayor elemento que hay en la colección.
- ✓ `E floor(E e)` Devuelve el mayor elemento de la colección que sea menor o igual que el elemento dado, o *null* si no existe tal elemento.
- ✓ `E ceiling(E e)` Devuelve el menor elemento de la colección que sea mayor o igual que el elemento dado, o *null* si no existe tal elemento.
- ✓ `E higher(E e)` Devuelve el menor elemento de la colección que sea estrictamente mayor al elemento dado, o *null* si no existe tal elemento.
- ✓ `E lower(E e)` Devuelve el mayor elemento de la colección que sea estrictamente menor que el elemento dado, o *null* si no existe tal elemento.
- ✓ `E pollFirst()` Recupera y elimina, el primer elemento (menor) de la colección o *null* si este conjunto está vacío.
- ✓ `E pollLast()` Recupera y elimina, el último elemento (mayor) de la colección o *null* si este conjunto está vacío.
- ✓ `SortedSet<E> headSet(E e)` Devuelve un subconjunto de la colección cuyos elementos son estrictamente menores que *e*. Los cambios en el subconjunto devuelto se reflejarán en el conjunto original y viceversa.
- ✓ `SortedSet<E> tailSet(E e)` Devuelve un subconjunto de la colección cuyos elementos son mayores o iguales que *e*. Los cambios en el subconjunto devuelto se reflejarán en el conjunto original y viceversa.
- ✓ `Iterator<E> descendingIterator()` Devuelve un iterador que permite recorrer los elementos de la colección en orden descendente.

La colección *TreeSet* almacena los elementos de forma ordenada en función de su valor. Cuando los elementos almacenados son de un tipo envoltorio, es fácil conocer su orden lexicográfico o numérico y realizar la ordenación, tal como se muestra en el siguiente ejemplo:

```
// TreeSet
TreeSet<Integer> setNombres=new TreeSet<>();
setNombres.add(3);
setNombres.add(1);
//setNombres.add(null);
setNombres.add(1);
setNombres.add(2);
// Recorriendo elementos
Iterator<Integer> itr=setNombres.iterator();
while(itr.hasNext()){
    System.out.println(itr.next());
}
// Imprimiendo todo el conjunto
System.out.println(setNombres);
```

```
run:
1
2
3
[1, 2, 3]
```

Sin embargo, si los elementos son de otros tipos la ordenación, puede estar realizándose de un modo difícilmente predecible. Para establecer cómo llevar a cabo la inserción de elementos de este tipo, es necesario construir la colección indicándole el comparador a emplear. Además, será necesario importar la clase *java.util.Comparator*. A continuación puede verse un ejemplo realizado con la clase *persona*:

```
// TreeSet
TreeSet<Persona> setPersonas=new TreeSet<>(Comparator.comparing(Persona::getNombre));
setPersonas.add(new Persona("María"));
setPersonas.add(new Persona("Ricardo"));
setPersonas.add(new Persona("Ricardo"));
//setNombres.add(null);
setPersonas.add(new Persona("Estefanía"));
setPersonas.add(new Persona("Javier"));
// Recorriendo elementos en orden creciente
System.out.println("Recorriendo en orden creciente:");
Iterator<Persona> itr=setPersonas.iterator();
while(itr.hasNext()){
    System.out.println(itr.next().getNombre());
}
// Recorriendo elementos en orden decreciente
System.out.println("Recorriendo en orden decreciente:");
Iterator<Persona> ditr=setPersonas.descendingIterator();
while(ditr.hasNext()){
    System.out.println(ditr.next().getNombre());
}
```

```
public class Persona {
    private String nombre;
    Persona (String nombre){
        this.nombre = nombre;
    }
    public String getNombre (){
        return nombre;
    }
}
```

```
run:
Recorriendo en orden creciente:
Estefanía
Javier
María
Ricardo
Recorriendo en orden decreciente:
Ricardo
María
Javier
Estefanía
```

Actividad propuesta 9.2

¿Qué comparador hace que la comparación se realice en orden inverso? ¿Qué comparador permite obviar si los caracteres de los elementos son mayúsculas o minúsculas?

9.2.2. Interfaz *List*

Representa un conjunto *lista* de elementos ordenados, que pueden estar duplicados y que permite acceso a los elementos a partir de la posición que ocupan. Existen varias colecciones que implementan la interfaz *List* y que pueden ser empleadas para codificar un *TAD Lista*. En todas se dispone de los métodos declarados en *Collection*.

A) ArrayList

Almacena los elementos en un *array dinámico* que va variando su tamaño en función del número de elementos a contener. Es la implementación de *List* más eficiente en la mayoría de los casos y permite valores *null*. El coste computacional de utilizar esta colección se dispara al eliminar o insertar elementos, ya que esto implica el desplazamiento de varios de ellos. Añade las funcionalidades siguientes:

- `boolean add(int posición, E elemento)` Añade el elemento que se le pasa por argumento en la posición indicada. Devuelve *true* si la colección se ha modificado.
- `boolean addAll(int posición, Collection<? extends E> c2)` Añade a la colección, en la posición indicada, todos los elementos de la colección *c2*. Retorna *true* si la colección original ha sido modificada.

- `E get(int posición)` Devuelve el elemento situado en la posición indicada. En caso de intentar acceder a elementos que no estén comprendidos en este rango el sistema lanzará la excepción *java.lang.IndexOutOfBoundsException*.
- `int indexOf(Object o)` Devuelve la posición de la primera ocurrencia del elemento dentro de la colección o -1 si no existe.
- `int lastIndexOf(Object o)` Devuelve la posición de la última ocurrencia del elemento dentro de la colección o -1 si no existe.
- `E remove(int posición)` Elimina el elemento situado en la posición indicada y lo devuelve.
- `E set(int posición, E elemento)` Sustituye el elemento situado en la posición especificada con el elemento que se pasa por argumento. La operación devuelve el elemento que ocupaba dicha posición.
- `sort(Comparator<? super E> c)` Ordena los elementos del *array*. Si los elementos no son de tipo envoltorio, o si se desea modificar la forma en la que realizar la ordenación, se puede utilizar un comparador. Otra alternativa es el empleo del método estático *sort* de la clase *Collections* que ordenará el *ArrayList* que reciba como argumento. Para poder hacer uso de él la primera tarea a realizar es importar *java.util.Collections*.
- `ListIterator<E> listIterator()` Devuelve un iterador que permite recorrer el *ArrayList* en ambos sentidos y modificar la lista durante el recorrido.

A continuación se muestra un ejemplo de declaración de un *ArrayList* unidimensional sobre el que posteriormente se accede a sus elementos:

```
ArrayList<Integer> arrayEnteros = new ArrayList<>();
arrayEnteros.add(22);
arrayEnteros.add(77);
arrayEnteros.add(11);
System.out.println("Primer elemento: "+arrayEnteros.get(0));
System.out.println("Segundo elemento: "+arrayEnteros.get(1));
Integer enteroReemplazado = arrayEnteros.set(1, 33);
System.out.println("Elemento reemplazado: "+enteroReemplazado);
System.out.println("Ahora segundo elemento: "+arrayEnteros.get(1));
System.out.println("Tercer elemento: "+arrayEnteros.get(2));
```

```
run:
Primer elemento: 22
Segundo elemento: 77
Elemento reemplazado: 77
Ahora segundo elemento: 33
Tercer elemento: 11
```

Otro ejemplo es el siguiente en el que se declara un *ArrayList* tridimensional sobre el que se van añadiendo elementos:

```
ArrayList<ArrayList<ArrayList>> array3D = new ArrayList<>();
array3D.add(new ArrayList<>());
array3D.add(new ArrayList<>());
array3D.get(0).add(new ArrayList<>());
array3D.get(0).add(new ArrayList<>());
array3D.get(1).add(new ArrayList<>());
array3D.get(1).add(new ArrayList<>());
array3D.get(1).add(new ArrayList<>());
array3D.get(0).get(0).add("cadena1");
array3D.get(0).get(1).add(true);
array3D.get(1).get(0).add("cadena2");
array3D.get(1).get(1).add(123);
array3D.get(1).get(2).add('Ç');
```

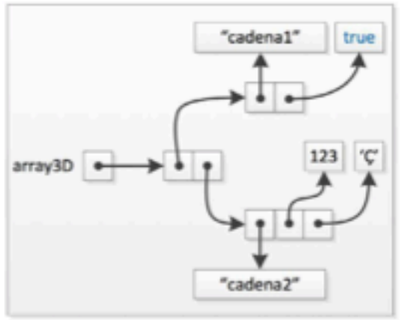


Figura 9.2
ArrayList tridimensional.

Un *ArrayList* podrá recorrerse de varias formas. Por ejemplo, utilizando un bucle *for* para recorrerlo tanto en orden incremental como decremental:

```
for (int i=0;i<arrayEnteros.size();i++){    //for incremental
    System.out.println(arrayEnteros.get(i));
}
for (int j=arrayEnteros.size();j>0;j--){    //for decremental
    System.out.println(arrayEnteros.get(j-1));
}
```

```
run:
22
77
11
11
77
22
```

La segunda alternativa es mediante un bucle *for each*:

```
for (int elemento:array){    //for each
    System.out.println(elemento);
}
```

```
arrayEnteros.forEach((elemento) -> { //for each Java 8
    System.out.println(elemento);
});
```

```
run:
22
77
11
```

Dado que la clase *ArrayList* implementa la interfaz *Iterable*, también es posible recorrer los elementos mediante un *iterador (iterator)*. Para ello el primer paso será importar *java.util.Iterator* y el recorrido se podrá realizar utilizando el método *next()* que retorna el siguiente elemento o mediante un bucle *for each*:

```
Iterator<Integer> iterador = arrayEnteros.iterator();
while (iterador.hasNext()) { // iterator
    System.out.println(iterador.next());
}
```

```
run:
22
77
11
```

```
Iterator<Integer> iterador = arrayEnteros.iterator();
iterador.forEachRemaining(elemento -> { // iterator y forEachRemaining
    System.out.println(elemento);
});
```

Para recorridos decrecientes emplearemos *listIterator*, que también permite recorridos en orden creciente. El recorrido se hace comprobando la existencia de elementos con *hasPrevious* y recuperándolos con *previous()*. Es necesario importar *java.util.ListIterator*:


```
ListIterator<Integer> listaIterador = arrayEnteros.listIterator(arrayEnteros.size());
while (listaIterador.hasPrevious()) { // listIterator
    Integer elemento = listaIterador.previous();
    System.out.println(elemento);
}
```

```
run:
11
77
22
```

Actividades propuestas

9.3. Otro método empleado para eliminar elementos es *removeIf*. Investiga sobre su uso y escribe un programa ejemplo en el que se utilice.

9.4. Busca información sobre cómo convertir un *array estático* en un *array dinámico* (*ArrayList*) y viceversa. Escribe el código fuente de un ejemplo que haga ambas conversiones.

9.5. ¿Qué método permite establecer de forma masiva y con una única llamada todos los elementos de una colección lista a un valor determinado? Propón un ejemplo de uso.

B) Vector

Similar a *ArrayList* pero garantiza accesos seguros (sincronizados) para su uso en aplicaciones multihilo. Aunque ha sido declarada obsoleta, es la clase padre que se describe a continuación ya que sigue siendo ampliamente utilizada.

C) Stack

Aun siendo una clase que implementa la interfaz *List*, en realidad representa al *TAD pila* en el que se utiliza *LIFO* como modo de acceso a los elementos (siempre se realiza por la parte superior, es decir, desde el último objeto apilado). Al ser una clase derivada de *Vector*, los accesos también son sincronizados, lo cual ralentiza las operaciones. Entre los métodos disponibles en esta clase se encuentran:

- ✓ `boolean empty()` Devuelve *true* si la pila está vacía.
- ✓ `E peek()` Devuelve el elemento situado en la cima de la pila sin eliminarlo de ella.
- ✓ `E pop()` Similar a *peek* pero eliminándolo de la pila ("desapilar").
- ✓ `E push(E item)` Añade un nuevo elemento en la cima de pila ("apilar").
- ✓ `int search(Object o)` Devuelve la distancia existente desde la cima de la pila al elemento pasado como argumento. Si el elemento no se encuentra en la pila, el resultado será -1, si está en la cima será 1, si está justo debajo de la cima 2… y así sucesivamente.

A continuación se muestra un ejemplo de implementación del *TAD pila* mediante *Stack*:

```
// Stack
Stack<String> st = new Stack();
st.push("Prueba");st.push("de");st.push("método");st.push("search");
st.push("en");st.push("una");st.push("cola");st.push("stack");
System.out.println("Posición 'cola': "+st.search("cola"));
```

```
run:
Posición 'cola': 2
```

Sabías que...

A pesar de llamarse *Stack (pila)*, su uso es altamente ineficiente para implementar estos TAD. En su defecto, es mejor utilizar colecciones que implementen la interfaz *Deque* o la colección *LinkedList* y dejar esta colección únicamente para cuando se necesite implementar este TAD sobre aplicaciones multihilo que necesiten accesos sincronizados.

D) LinkedList

La colección *LinkedList* es la implementación de las interfaces *List* y *Deque*. Se compone de una lista doblemente enlazada en la que cada elemento almacena una referencia a los elementos siguiente y anterior. Entre sus características principales están:

- El almacenamiento de sus elementos en memoria, a diferencia del *array*, no es contiguo.
- Permite almacenar valores *null*.
- El coste de acceder a un elemento aleatorio se dispara puesto que es necesario recorrer todos los anteriores o posteriores hasta posicionarse en él. Sin embargo, una vez se tiene la referencia del elemento, el coste por añadir o insertar es menor al realizarse mediante un cambio de referencias.
- Se trata de la colección ideal para implementar los *TAD pila* y *cola*.

Entre los métodos disponibles en esta clase, se encuentran todos los descritos anteriormente para *ArrayList* más los que le permiten llevar a cabo la implementación de pilas y colas (cuadro 9.1).

Cuadro 9.1
Métodos de *LinkedList* más destacados

`addFirst(E e)`
`boolean offerFirst(E e)`
`push(E e)`
	Inserta el elemento *e* al comienzo de la lista.

`addLast(E e)`
`boolean offer(E e)`
`boolean offerLast(E e)`
	Inserta el elemento *e* al final de la lista.

`E element()`
`E getFirst()`
`E peek()`
`E peekFirst()`
	Devuelve el elemento situado al comienzo de la lista sin eliminarlo de ella.

[.../...]

Cuadro 9.1 (cont.)

`E poll()`
`E pollFirst()`
`E pop()`
`E remove()`
`E removeFirst()`
	Devuelve y elimina el elemento situado al comienzo de la lista.

`E getLast()` `E peekLast()`	Devuelve el elemento situado al final de la lista sin eliminarlo de ella.
`E pollLast()` `E removeLast()`	Devuelve y elimina el elemento situado al final de la lista.
`Iterator<E> descendingIterator()`	Devuelve un iterador que permite recorrer los elementos en orden descendente.
`boolean removeFirstOccurrence(Object o)`	Elimina la primera ocurrencia del elemento que se le pasa por argumento.
`boolean removeLastOccurrence(Object o)`	Elimina la última ocurrencia del elemento que se le pasa por argumento.

1. TAD Cola

Una *cola* es un tipo abstracto de datos que permite almacenar y recuperar información siguiendo un esquema *FIFO*. Toda cola debe contar, al menos, con las operaciones *encolar* (*add*) y *desencolar* (*remove*). En todo momento se tendrá acceso al elemento que más tiempo lleva almacenado (denominado *frente* de cola).

A continuación se muestra un ejemplo básico de implementación de *cola* utilizando la colección *LinkedList*:

```
package cola;
import java.util.LinkedList;

public class Cola {
    private final LinkedList cola;

    public Cola() {this.cola = new LinkedList();}
    public void encolar(Object elemento) {cola.add(elemento);}
    public Object desencolar() {return cola.remove();}
    public Object frente() {return cola.peek();}

    public static void main(String[] args) {
        Cola c = new Cola();
        System.out.println("Elemento frente:" + c.frente());
        for (int i=1;i<=3;i++)
            c.encolar(i);
        System.out.println("Elemento frente:" + c.frente());
        c.desencolar();
        System.out.println("Elemento frente:" + c.frente());
    }
}
```

```
run:
Elemento frente:null
Elemento frente:1
Elemento frente:2
```

Figura 9.3
Funcionamiento TAD Cola.

2. TAD Pila

Como ya se ha explicado anteriormente, una *pila* es un tipo abstracto de datos que permite almacenar y recuperar información siguiendo un esquema *LIFO*. Toda pila debe contar, al menos, con las operaciones *apilar* (*push*) y *desapilar* (*pop*). En todo momento se tendrá acceso a la cima de la pila en la que estará el último elemento apilado, también conocido como *TOS*.

Figura 9.4
Funcionamiento TAD Pila.

A continuación se muestra un ejemplo básico de implementación de *pila* utilizando la colección *LinkedList*:

```
package pila;
import java.util.LinkedList;

public class Pila {
    private final LinkedList pila;

    public Pila() {this.pila = new LinkedList();}
    public void apilar(Object elemento) {pila.push(elemento);}
    public Object desapilar() {return pila.pop();}
    public Object cima() {return pila.peek();}

    public static void main(String[] args) {
        Pila p = new Pila();
        for (int i=1;i<=3;i++)
            p.apilar(i);
        System.out.println("Elemento TOS:" + p.cima());
        p.apilar(4);
        System.out.println("Elemento TOS:" + p.cima());
        p.desapilar();
        System.out.println("Elemento TOS:" + p.cima());
    }
}
```

```
run:
Elemento TOS:3
Elemento TOS:4
Elemento TOS:3
```

Investiga

Busca información sobre los métodos *add* y *offer* que permiten añadir elementos a la colección *LinkedList* y describe las diferencias existentes entre ellos.

9.2.3. Interfaz *Queue*

También conocida como *cola*, representa otra estructura de almacenamiento *FIFO*. Existen varias implementaciones de esta interfaz que permiten implementar *colas* cuyo comportamiento difiere al de las implementaciones vistas anteriormente.

A) PriorityQueue

Los elementos de una cola de prioridad son ordenados de acuerdo con su orden natural independientemente del momento en el que fuesen encolados, aunque también es posible emplear un comparador que es pasado como argumento cuando se construye la cola. Este tipo de colas no permiten elementos nulos ni la inserción de objetos no comparables, aunque sí duplicados. Al igual que con la colección *TreeSet*, es posible *customizar* el comparador a emplear.

Algunos de los métodos de esta colección son *add*, *clear*, *contains*, *iterator*, *offer*, *peek*, *poll*, *remove* y *size*. Su sintaxis de uso y función es similar a las descritas en colecciones anteriores, por lo que no será descrita en detalle y se mostrará un simple ejemplo de uso:

```
PriorityQueue<Integer> numeros = new PriorityQueue<>();
// Encolando elementos
numeros.add(22);numeros.add(11);numeros.add(30);
numeros.add(25);numeros.add(11);numeros.add(6);
// Desencolando elementos según han sido ordenados
while (!numeros.isEmpty()) {
    System.out.println(numeros.remove());
}
```

```
run:
6
11
11
22
25
30
```

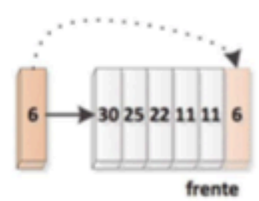


Figura 9.5
Funcionamiento TAD Cola de prioridad.

B) ArrayDeque

Es un tipo de *array* que implementa la subinterfaz *Deque* de la interfaz *Queue*. Utiliza una matriz de tamaño variable y no admite la inserción de elementos nulos. Esta clase y su iterador implementan todos los métodos de las interfaces *Collection* e *Iterator*. Puede ser utilizada como *pila*, como *cola* o como ambos a la vez, dado que permite insertar elementos por ambos extremos. Es posible consultar la lista de métodos de la colección *LinkedList* como referencia para *ArrayDeque*. A continuación, se muestra un ejemplo de uso:

```
ArrayDeque<String> nombres = new ArrayDeque<>();
// Añadiendo elementos
nombres.add("Ana");       // Ana
nombres.add("Zoe");       // Ana Zoe
nombres.push("Luis");     // Luis Ana Zoe
nombres.add("Luz");       // Luis Ana Zoe Luz
nombres.push("Piero");    // Piero Luis Ana Zoe Luz
nombres.push("Vanesa");   // Vanesa Piero Luis Ana Zoe Luz
while (!nombres.isEmpty()) {
    System.out.println(nombres.pop());
}
```

```
run:
Vanesa
Piero
Luis
Ana
Zoe
Luz
```

Actividades propuestas

9.6. Tanto *LinkedList* como *ArrayDeque* pueden ser utilizados como colas, ¿cuál es, en términos generales, más rápida de las dos?

9.7. Tanto *Stack* como *ArrayDeque* pueden ser utilizados como pilas, ¿cuál es, en términos generales, más rápida de las dos?

9.3. Interfaz *Map*

La interfaz *Map* permite almacenar pares clave-valor de manera que sea fácil buscar un valor a partir de su clave. Las colecciones que implementan *Map* no pueden contener claves duplicadas. Cada clave únicamente se corresponderá con un valor. A continuación, se enumeran y describen los métodos que todas las colecciones que implementan *Map* tienen disponible:

- ✓ `void clear()` Elimina todos los pares clave-valor del mapa.
- ✓ `boolean containsKey(Object clave)` Devuelve *true* si la clave existe en el mapa y *false* en caso contrario.
- ✓ `boolean containsValue(Object valor)` Devuelve *true* si existe al menos una clave en el mapa asociada a dicho valor y *false* en caso contrario.
- ✓ `set<Map.Entry<K,V>> entrySet()` Devuelve un conjunto que contiene todos los pares clave-valor del mapa.
- ✓ `Object get(Object clave)` Devuelve el valor asociado a la clave.
- ✓ `boolean isEmpty()` Devuelve *true* si el mapa no contiene ningún par clave-valor.

- ✓ `set<K> keySet()` Devuelve un conjunto que contiene todas las claves del mapa.
- ✓ `Object put(Object clave, Object valor)` Inserta una entrada clave-valor en el mapa. Devuelve el valor anteriormente asociado a clave o *null* si no existía.
- ✓ `void putAll(Map m)` Copia el mapa *m* al mapa actual.
- ✓ `Object remove(Object clave)` Elimina del mapa el par clave-valor correspondiente a la clave pasada y lo devuelve.
- ✓ `boolean remove(Object clave, Object valor)` Elimina del mapa el par clave-valor cuya clave y valor coinciden con los indicados. Si lo elimina devuelve *true*.
- ✓ `V replace(K clave, V valor)` Sustituye el valor del par correspondiente a la clave pasada por argumento. Devuelve el valor anterior si se ha realizado algún cambio, o *null* en caso contrario.
- ✓ `boolean replace(K clave, V valorOriginal, V nuevoValor)` Para la clave pasada por argumento, si su valor es *valorOriginal* se sustituye por *nuevoValor*. Devuelve *true* si se ha realizado algún cambio.
- ✓ `int size()` Devuelve el número de elementos contenidos en el mapa.
- ✓ `Collection<V> values()` Devuelve una colección con los valores almacenados en el mapa. Los cambios realizados en la colección se reflejarán en el mapa original y viceversa.

9.3.1. *HashMap*

Almacena las claves en una tabla *hash* admitiendo incluso una entrada *null*. Es la implementación con mejor rendimiento de todas, pero no garantiza ningún orden a la hora de realizar iteraciones. Puede ser interesante definir el tamaño inicial de la tabla al llamar al constructor, ya que esto puede afectar en el rendimiento. A continuación, se puede ver un ejemplo de uso:

```
HashMap<Integer, String> hashMap = new HashMap<>();
hashMap.put(1, "1");
hashMap.put(3, "3");
hashMap.put(4, "4");
hashMap.put(2, "2");
hashMap.put(5, "5");
System.out.println(hashMap.values());
```

```
run:
[1, 2, 3, 4, 5]
```

9.3.2. *LinkedHashMap*

Esta implementación se diferencia de *HashMap* en que mantiene una lista con un enlace doble en todas sus entradas. Debido a esto es un poco más costosa, pero se define el orden de iteración, que normalmente es el orden en que se insertaron las claves en el mapa. El orden de inserción no se verá afectado si una clave se reinserta en el mapa. Puede conocerse mejor cómo se comporta *LinkedHashMap* a través del siguiente ejemplo:

```
LinkedHashMap<Integer, String> linkedHashMap = new LinkedHashMap<>();
linkedHashMap.put(1, "1");
linkedHashMap.put(3, "3");
linkedHashMap.put(4, "4");
linkedHashMap.put(2, "2");
linkedHashMap.put(5, "5");
System.out.println(linkedHashMap.values());
```

```
run:
[1, 3, 4, 2, 5]
```

9.3.3. *TreeMap*

Es bastante más lento que *HashMap*, ya que almacena las claves ordenadas ascendentemente en un árbol. Las claves deben implementar la interfaz *Comparable*. En esta colección no se admiten valores *null*. Además de los métodos de la interfaz *Map*, implementa:

- `Map.Entry<K,V> ceilingEntry(K clave)` Devuelve el par clave-valor asociado a la menor clave de todas aquellas claves que son mayores o igual a la pasada por argumento; devuelve *null* si no existe.
- `Map.Entry<K,V> higherEntry(K clave)` Idéntico a *ceilingEntry* pero para las claves que son estrictamente mayores a la pasada por argumento.
- `Map.Entry<K,V> firstEntry()` Devuelve el par clave-valor asociado con la menor clave del mapa o *null* si el mapa está vacío.
- `Map.Entry<K,V> floorEntry(K clave)` Devuelve un par clave-valor asociado con la mayor clave de todas las claves menores o iguales a la pasada por argumento; devuelve *null* si no existe.
- `Map.Entry<K,V> lastEntry()` Similar a *firstEntry* pero para la mayor clave del mapa.
- `K ceilingKey(K clave)` Devuelve la menor clave de todas aquellas claves que sean mayores o igual a la pasada por argumento; devuelve *null* si no existe.
- `K higherKey(K clave)` Idéntico a *ceilingKey* pero para las claves que sean estrictamente mayores a la pasada por argumento.
- `K firstKey()` Devuelve la menor clave del mapa.
- `K floorKey(K key)` Devuelve la mayor clave de todas aquellas claves que sean menores o iguales a la pasada por argumento; devuelve *null* si no existe.
- `K lastKey()` Devuelve la mayor clave del mapa.
- `SortedMap<K,V> headMap(K clave)` Devuelve un submapa de pares cuya clave es estrictamente menor a la clave pasada como argumento. Los cambios realizados en el submapa se reflejarán en el mapa original y viceversa.
- `NavigableMap<K,V> headMap(K clave, true)` Similar al anterior pero incluyendo también, en caso de existir, el par correspondiente a la clave pasada como argumento.
- `SortedMap<K,V> tailMap(K clave)` Devuelve un submapa de pares cuya clave es mayor o igual a la clave pasada como argumento. Los cambios realizados en el submapa se reflejarán en el mapa original y viceversa.
- `NavigableMap<K,V> tailMap(K clave, true)` Similar al anterior pero incluyendo también, en caso de existir, el par correspondiente a la clave pasada como argumento.

A continuación, se muestra un ejemplo de uso de *TreeMap*:

```
TreeMap<Integer,String> puntuaciones =new TreeMap<>();
puntuaciones.put(8,"Notable alto");
puntuaciones.put(5,"Aprobado");
puntuaciones.put(10,"Matrícula H.");
puntuaciones.put(6,"Bien");
puntuaciones.put(9,"Sobresaliente");
puntuaciones.put(7,"Notable");
puntuaciones.entrySet().forEach((m) -> {
    System.out.println(m.getKey()+" "+m.getValue());
});
puntuaciones.replace(7, "Notable", "Notable bajo");
System.out.println(puntuaciones.values());
System.out.println("Por debajo de "+puntuaciones.firstKey()+" es suspenso");
```

```
run:
5 Aprobado
6 Bien
7 Notable
8 Notable alto
9 Sobresaliente
10 Matrícula H.
[Aprobado, Bien, Notable bajo, Notable alto, Sobresaliente, Matrícula H.]
Por debajo de 5 es suspenso
```

Para saber más

Cualquier objeto colección puede utilizarse como tipo del argumento enviado a un método o ser el tipo del valor de su retorno.

```
public ArrayList<Integer> ejemplo (ArrayList<String> array){
    ArrayList<Integer> nombreArrayList = new ArrayList<>();
    // ...
    return nombreArrayList;
}
```

Actividad propuesta 9.8

Realiza un estudio comparativo entre la clase *Hashtable* y *HashMap*.

Resumen

- Los conjuntos de elementos pueden ser manejados fácilmente con colecciones que implementen la interfaz *Set*. En un conjunto no puede haber un elemento duplicado.
- El *TAD Lista* se caracteriza por tener una lista de elementos, ordenada o no, y puede desarrollarse fácilmente con cualquier colección que implemente la interfaz *List*. En una lista se podrá recuperar cualquier elemento.
- El *TAD Cola* se caracteriza por tener un grupo de elementos de los que únicamente se puede recuperar uno que es conocido como *frente* de cola. Si el orden de los elementos de la cola viene dado por el momento en que fueron insertados, se tendrá una cola *FIFO*. Esta puede desarrollarse fácilmente con la colección *LinkedList* o *ArrayDeque*, aunque también puede utilizarse *ArrayList*. Si se trata de una cola en la que el orden es dado por el valor de los elementos a encolar, se tendrá una cola por prioridades para la que es mejor utilizar *PriorityQueue*.

- El *TAD Pila* se caracteriza por tener un grupo de elementos de los que únicamente se puede recuperar uno que es conocido como *cima* de pila y que siempre es el último que se ha insertado. Una *pila* trabaja según un esquema *LIFO*. Esta puede desarrollarse fácilmente con la colección *LinkedList* o *ArrayDeque*, aunque también puede utilizarse *ArrayList*. Si se necesita seguridad en los accesos, es mejor emplear *Stack*.
- Los mapas son conjuntos de elementos clave que están asociados a otro elemento valor. Son representados mediante colecciones que implementan la interfaz *Map*. En un mapa cada clave es única; no obstante, puede haber claves diferentes con mismo valor.

Ejercicios propuestos

1. Describe la diferencia existente entre colecciones *Set* y *Map*.
2. ¿Es posible implementar una pila utilizando colecciones *Map*? ¿Y una cola de prioridades? Justifica tu respuesta.
3. ¿Qué diferencias existen entre un *array* y un *ArrayList*?
4. ¿Es correcta la siguiente instrucción: *Collection<Object>c = new TreeSet<>();*? Justifica tu respuesta.
5. ¿Cuál de las colecciones vistas en el capítulo no se recomienda utilizar salvo en aplicaciones multihilo?
6. Diseña una tabla resumen en la que se recojan las principales colecciones expuestas en el capítulo y sus características más significativas (ordenada o no, permite duplicados o no, acceso secuencial o aleatorio…).
7. Elabora un diagrama de flujo que, mediante el empleo de preguntas, te vaya guiando hasta el tipo de colección más adecuado para cada situación. Las preguntas pueden ser del tipo: "¿Debe mantener el orden?", "¿Contendrá *null*?", "¿Admite duplicados?"…

Supuestos prácticos

1. Construye un programa que determine cuál de las 3 colecciones conjunto vistas en el capítulo (*HashSet*, *LinkedHashSet* o *TreeSet*) es más rápida a la hora de añadir elementos al conjunto, comprobar si un elemento existe en el conjunto y eliminar un elemento del conjunto. (Mide el tiempo antes y después de las operaciones con *System.currentTimeMillis()*).

2. Se necesita desarrollar un programa que gestione los pedidos que llegan a un almacén desde las diferentes tiendas a las que se suministra. El suministro se realiza por riguroso orden de llegada de cada pedido y bajo ningún concepto se puede trabajar en un pedido sin haber finalizado el correspondiente. Cada pedido contendrá información sobre el cliente que lo ha realizado y los productos que se han solicitado (tipo y cantidad). Obviamente los mismos productos pueden estar en varios pedidos y cualquier cliente puede solicitar los mismos productos en diferentes pedidos. Implementa el TAD más apropiado para dar solución a esta situación, elige las colecciones que empleará para ello y desarrolla las clases pedido, producto y cliente.
3. Desarrolla un programa que lea constantemente desde la entrada estándar hasta recibir un *enter* como finalizador. Cada carácter será almacenado de forma independiente en una colección que permitirá recuperar el contenido en orden inverso al que fueron introducidos. Una vez recibido *enter*, recorre la colección mostrando por pantalla el contenido almacenado. Recuerda que los caracteres deben leerse por separado y no se debe emplear la clase *String*.
4. A partir del programa del ejercicio anterior, añade la funcionalidad necesaria para que también se muestren ordenados lexicográficamente y sin repeticiones los caracteres que se han introducido.
5. Se desea desarrollar un programa que permita gestionar la información de los alumnos de una clase. El programa comenzará pidiendo el número de alumnos a gestionar y, acto seguido, pedirá los datos de los alumnos. De cada alumno se almacenará *apellido1*, *apellido2*, *nombre* y *nacionalidad* en objetos de la clase *alumno* que también deberá desarrollar. No es relevante de qué alumno se introduzcan antes los datos, pero sí que queden almacenados en el orden establecido por *apellido1*, *apellido2* y *nombre*. Una vez recogidos los datos de todos los alumnos, el programa mostrará un resumen ordenado de los datos recogidos. Implementa el programa indicado seleccionando la colección que más se ajuste a lo requerido.

Actividades de autoevaluación

1. ¿Qué método devuelve *false* si una pila contiene elementos?
   - ☐ isEmpty()
   - ☐ clear()
   - ☐ empty()
   - ☐ size()

2. Para recorrer los elementos de una *LinkedList* en orden decreciente emplearemos:
   - ☐ *a)* *Iterator<E> iterator()*
   - ☐ *b)* *Iterator<E> descendingIterator()*
   - ☐ *c)* *ListIterator<E> listIterator()*
   - ☐ *d)* No es posible recorrer en orden decreciente una *LinkedList*.

3. ¿Cuál de las siguientes afirmaciones es verdadera?
   - ☐ *a)* En una colección *Map* nunca se pueden introducir elementos *null*.
   - ☐ *b)* En una colección *List* no se pueden duplicar elementos.
   - ☐ *c)* En una colección *Set* no es posible almacenar elementos clave-valor.
   - ☐ *d)* Todas las opciones anteriores son incorrectas.

4. ¿Cuál de las siguientes colecciones no permite implementar una *cola*?
   - ☐ *a)* *LinkedList*
   - ☐ *b)* *PriorityQueue*
   - ☐ *c)* *ArrayDeque*
   - ☐ *d)* *TreeSet*

5. Identifica cuál de las siguientes colecciones ofrece seguridad para aplicaciones multihilo:
   - ☐ *a)* *LinkedHashMap*
   - ☐ *b)* *ArrayDeque*
   - ☐ *c)* *Stack*
   - ☐ *d)* *LinkedList*

6. ¿Cuál de las siguientes colecciones ofrece seguridad para aplicaciones multihilo y es similar a *ArrayList* en cuanto a funcionalidad se refiere?
   - ☐ *a)* *Vector*
   - ☐ *b)* *ArrayDeque*
   - ☐ *c)* *Stack*
   - ☐ *d)* *LinkedList*

7. ¿Es posible utilizar colecciones para almacenar elementos que sean otras colecciones?
   - ☐ *a)* No.
   - ☐ *b)* Sí, pero solo con *ArrayList* para hacer *arrays* dinámicos de varias dimensiones.
   - ☐ *c)* Sí, pero solo si son del mismo tipo.
   - ☐ *d)* Sí.

8. En una cola de prioridades:
   - ☐ *a)* Los elementos se ordenan según sus valores.
   - ☐ *b)* Es posible modificar la forma en la que se ordenarán.
   - ☐ *c)* Las opciones *a)* y *b)* son verdaderas.
   - ☐ *d)* Todas las opciones anteriores son incorrectas.

9. Identifica la afirmación falsa:
   - ☐ *a)* La colección *HashSet* almacena los elementos en una colección *HashMap*.
   - ☐ *b)* En una *LinkedHashSet* los elementos son ordenados según han sido insertados.
   - ☐ *c)* La colección *TreeSet* almacena los elementos en orden ascendente según sus valores.
   - ☐ *d)* Las colecciones *TreeSet*, *HashSet* y *LinkedHashSet* permiten valores *null*.

10. El único elemento accesible en una pila es conocido como:
   - ☐ *a)* Raíz.
   - ☐ *b)* Frente.
   - ☐ *c)* TOS o cima de pila.
   - ☐ *d)* Todas las opciones anteriores son incorrectas.

SOLUCIONES:

1.	a	b	**c**	d	5.	a	b	**c**	d						
2.	a	**b**	c	d	6.	**a**	b	c	d	9.	a	b	c	**d**	
3.	a	b	**c**	d	7.	a	b	c	**d**	10.	a	b	**c**	d	
4.	a	b	c	**d**	8.	a	b	**c**	d						

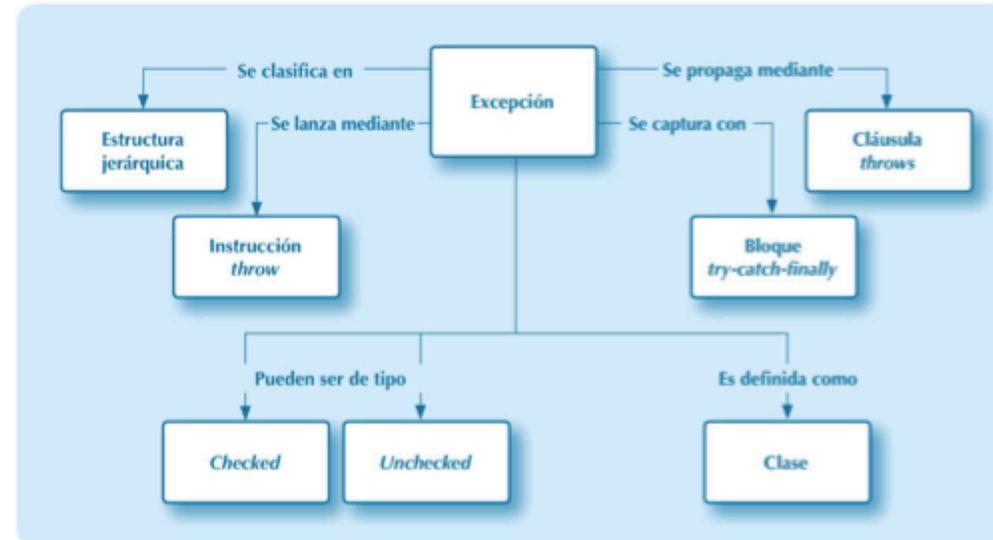
10

Control y manejo de excepciones

Objetivos

- ✓ Conocer el concepto de *excepción* y familiarizarse con su uso.
- ✓ Distinguir entre los diferentes tipos de excepciones y saber clasificarlos.
- ✓ Aprender a capturar las posibles excepciones que se produzcan en un programa.
- ✓ Dominar el mecanismo para propagar excepciones a un nivel superior.
- ✓ Saber cómo lanzar excepciones de forma manual.
- ✓ Diseñar excepciones personalizadas para atender a diferentes tipos de error.

Mapa conceptual



Glosario

Ad hoc. Expresión que se refiere a una solución generada específicamente para un determinado fin.

API. También conocida como *application programming interface* o interfaz de programación de aplicaciones, es un conjunto de funciones y procedimientos que son ofrecidas para ser utilizadas evitando así la construcción de todo el código.

Checked. Tipo de excepción que es controlada en tiempo de compilación y que permite recuperar el programa. Es obligatorio controlarlas pues, de lo contrario, no se generará el código intermedio necesario para la ejecución del programa.

Excepción. Evento producido durante la ejecución de un programa que altera el flujo normal de ejecución.

Pila de llamadas. Estructura de datos *LIFO* que almacena información sobre las diferentes llamadas a los métodos de un programa. Permite conocer la interacción de unos métodos con otros y depurar programas ante posibles errores.

Traza. Secuencia de instrucciones que se han ejecutado hasta un determinado punto del programa.

Unchecked. Tipo de excepción, generalmente producido por errores ajenos al código del programa que se está ejecutando, que no es obligatorio controlar.

10.1. Excepciones

A lo largo de los capítulos anteriores se ha dado por hecho en todo momento que los resultados proporcionados por los diferentes métodos siempre han sido los esperados, pero esto no siempre ocurre así. En ocasiones se producen errores en tiempo de ejecución que, de no ser tenidos en cuenta y gestionarse correctamente, provocarán una finalización errónea del programa.

A este tipo de eventos producidos por los errores y que se dan durante la ejecución del programa alterando el flujo normal de ejecución es a lo que se le denomina *excepciones*.

Figura 10.1
Lanzamiento, propagación y captura de excepciones.

Llevar un buen control de las excepciones permite:

- Separar el código destinado a controlar errores del resto del código normal y diseñado para un fin concreto.
- Propagar los errores sobre la pila de llamadas.
- Agrupar los diferentes tipos de errores y proponer códigos alternativos para tratarlos.

10.2. Jerarquía de excepciones

Java pone a disposición del programador un conjunto muy completo de diferentes tipos de excepciones con las que controlar las situaciones que puedan provocar esos errores inesperados.

Todas las excepciones en Java se manejan mediante objetos de clases que heredan, en última instancia, de la clase *java.lang.Throwable*, siendo cada una de ellas de un tipo concreto. En la figura 10.2 puede verse la jerarquía y tipos de excepciones en Java.

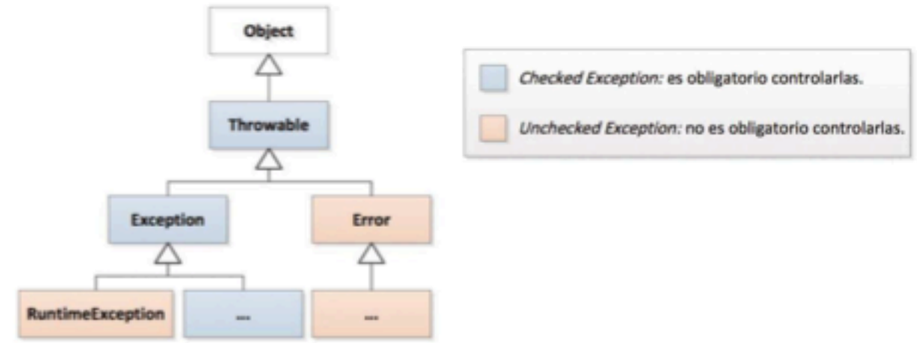


Figura 10.2
Jerarquía y tipo de excepciones.

Todas las excepciones anteriores se pueden clasificar en:

1. *Checked:* representan errores de los que se puede recuperar el programa. Todas estas excepciones deben ser capturadas y manejadas convenientemente en tiempo de compilación.
2. *Unchecked:* representan errores de programación. Estas excepciones no deben ser forzosamente declaradas ni capturadas, aunque de no hacerlo es posible que el programa finalice erróneamente. A continuación, se muestra un ejemplo de este tipo de excepción que se da al acceder a posiciones erróneas de un *array* y que podría haberse evitado fácilmente con una simple comprobación previa del valor de la variable *posicionAcceso*:

```
int[] arrayTresElementos = {4, 1, 7}; // Array de 3 elementos
int posicionAcceso = 3;
System.out.println("El 4º elemento es: "+arrayTresElementos[posicionAcceso]);
```

```
run:
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3
        at excepciones.Excepciones.main(Excepciones.java:5)
```

Throwable y cualquier subclase de *Throwable* que no sea también una subclase de *RuntimeException* o *Error* se consideran excepciones *Checked*. El resto son consideradas *Unchecked*.

Recurso web

Existe un amplio catálogo de excepciones que heredan de *Exception* y *Error* y que no han sido mencionadas. En la URL https://docs.oracle.com/javase/8/docs/api/ podrás consultar la API de Java para conocer más sobre ellas.

Actividades propuestas

10.1. ¿Qué tipo de excepciones representa `RuntimeException`?

10.2. ¿Qué tipo de excepción se producirá al ejecutarse `int n = Integer.parseInt("a");`?

10.3. Miembros de una excepción

Toda excepción, por el hecho de ser un objeto de una clase, describe un conjunto de miembros que permiten hacer uso de ella. Se dispondrá de varios métodos constructor para poder instanciarlo vacío, con un mensaje concreto de error o con una causa de excepción.

En cuanto a los métodos disponibles, es importante recordar que algunos de ellos son heredados de la clase *Object* mientras que otros son específicos de objetos excepción. En el cuadro 10.1 pueden verse los más destacables.

CUADRO 10.1
Métodos de las clases que representan excepciones

`String toString()`	Devolverá una cadena formada por: *TipoExcepción:* mensaje. Siendo *mensaje* el texto con el que se construyó la excepción.
`Class<?> getClass()`	Devuelve la clase a la que pertenece la excepción.
`String getMessage()`	Devuelve el mensaje con el que fue creada la excepción.
`printStackTrace()`	Imprime por la salida de error el objeto desde el que se invoca junto con una traza con las llamadas a los métodos desde los que se ha producido la excepción. Es muy útil para depurar programas, ya que permite conocer el punto exacto en el que se produjo el error.

10.4. Manejo de excepciones

Todo método con código susceptible de generar excepciones debe incluir los bloques necesarios para capturarlas o, en su defecto, declararlas en su firma para que cuando se vaya a invocar a dicho método se sepa que es capaz de generar una excepción. A continuación, se mostrarán las distintas alternativas para el manejo de excepciones con ejemplos ilustrativos de su uso.

10.4.1. Captura de excepciones

Se realiza mediante bloques *try-catch-finally* conforme a:

1. En primer lugar el bloque *try*, que encerrará entre llaves el código susceptible de generar una excepción:

```
try {
    ... // código susceptible de generar excepciones
}
```

2. A continuación se incluirán tantos bloques *catch* como diferentes tipos de excepción a capturar con el código necesario para gestionar esos errores. De esta manera, ante errores que provoquen excepciones de esos tipos siempre habrá un código encargado de gestionar el error y reconducir de forma controlada el flujo de ejecución. Es importante tener en cuenta que, al existir relaciones de herencia entre los diferentes tipos de excepción, es posible que la declaración de un bloque *catch* para una excepción más genérica (padre) capture excepciones de clases derivadas (hijas) si no se han definido bloques *catch* más específicos para ellas. En este ejemplo se declaran 2 bloques de captura de excepciones de diferente tipo:

```
try {
    ... // código susceptible de generar excepciones
} catch (Tipo1 excepción1){
    ... // código encargado de gestionar excepciones Tipo1
} catch (Tipo2 excepción2){
    ... // código encargado de gestionar excepciones Tipo2
}
```

Cada bloque *try* podrá anidarse dentro de otro. De esta manera cada excepción generada dentro del bloque *try* interno que no sea capturada por el bloque *catch* asociado podrá ser capturada por el bloque *catch* externo:

```
try { // try externo
    ...
    try { // try anidado
        ...
    } catch (Tipo1 excepción1) {
        // Capturando la excepción
        ...
    }
} catch (Tipo2 excepción2) {
    // Capturando la excepción
    ...
}
```

Desde la versión Java SE 7 es posible realizar la captura de dos o más excepciones en la misma cláusula *catch* en lo que se conoce como *multicatch*, lo cual puede ser muy útil en aquellos casos en los que el código a ejecutar ante excepciones diferentes sea el mismo. Cada tipo de excepción se declara separado del resto por el carácter barra (|). Ejemplo:

Bloques equivalentes

```
catch (Tipo1 ex) {
    // Capturando la excepción
    ...
} catch (Tipo2 ex) {
    // Capturando la excepción
    ...
}
```

```
catch (Tipo1|Tipo2 ex) {
    // Capturando la excepción
    ...
}
```

3. Por último, y opcionalmente, podrá declararse un bloque *finally*. Si el bloque *try* asociado comienza su ejecución, el bloque *finally* siempre se ejecutará y lo hará:

   a) Justo detrás del bloque *try* si no se produjo ninguna excepción.
   b) Justo detrás del bloque *catch* correspondiente si se produjera alguna excepción y esta se capturase.
   c) Inmediatamente después de que surja la excepción si esta no es capturada por ningún bloque *catch* existente. A continuación, la excepción se propagará hacia arriba.

   La sintaxis completa de declaración de código de control de excepciones es:

```
try {
   ... // código susceptible de generar excepciones
} catch (Tipo1 excepción1){
   ... // código encargado de gestionar excepciones Tipo1
} finally {
   ... // código que siempre se ejecutará
}
```

En el siguiente ejemplo podrá comprobarse cómo se captura una excepción y cómo se comporta el bloque *finally* conforme a las descripciones anteriores:

```
System.out.println("COMIENZO PROGRAMA DIVISIÓN");
try{
    Scanner entradaEscaner = new Scanner (System.in);
    System.out.println("Introduzca dividendo:");
    int dividendo = Integer.parseInt(entradaEscaner.nextLine ());
    System.out.println("Introduzca divisor:");
    int divisor = Integer.parseInt(entradaEscaner.nextLine ());
    System.out.println("Resultado :"+dividendo/divisor);
}catch (java.lang.ArithmeticException ex){
    System.out.println("Imposible dividir por 0");
}finally{
    System.out.println("FIN PROGRAMA DIVISIÓN");
}
```


A

```
run:
COMIENZO PROGRAMA DIVISIÓN
Introduzca dividendo:
10
Introduzca divisor:
2
Resultado :5
FIN PROGRAMA DIVISIÓN
```

B

```
run:
COMIENZO PROGRAMA DIVISIÓN
Introduzca dividendo:
10
Introduzca divisor:
0
Imposible dividir por 0
FIN PROGRAMA DIVISIÓN
```

C

```
run:
COMIENZO PROGRAMA DIVISIÓN
Introduzca dividendo:
DIEZ
FIN PROGRAMA DIVISIÓN
Exception in thread "main" java.lang.NumberFormatException: For input string: "DIEZ"
        at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
        at java.lang.Integer.parseInt(Integer.java:580)
```

Recurso web

La declaración *try-with-resources* es una declaración *try* que declara uno o más recursos que serán utilizados en el bloque y que serán cerrados automáticamente al final de la declaración. Los recursos a incluir deben ser objetos que implementen *java.lang.AutoCloseable*. Está disponible a partir de Java SE 7. Para conocer más sobre ella consulta la URL https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html.

Actividades propuestas

10.3. Describe el comportamiento del último código del apartado 10.4.1 y la salida que se hubiese obtenido si se añadiera un bloque *catch* justo a continuación del bloque *try* para capturar excepciones del tipo *java.lang.Exception* y se repitieran las mismas entradas.

10.4. ¿Es posible capturar excepciones de tipo *Throwable*? Justifica tu respuesta.

10.4.2. Propagación de excepciones

Otra de las alternativas que ofrece Java para el control de excepciones es *propagar* la excepción hacia el método desde el cual se realizó la invocación para que sea allí donde se capture.

Para ello será necesario declarar en la firma del método que este puede generar excepciones de un tipo concreto. Esto se hace añadiendo al final del método la cláusula *throws* seguida de un listado de las excepciones que podrán ser relanzadas:

```
[visibilidad] [static] tipo nombre ([tipo parám.1] … [,tipo parám.N])
throws tipoExcepcion1, tipoExcepcion2, …, tipoExcepcionN {…}
```

En el siguiente ejemplo puede verse cómo se ha anunciado que dentro del método *leerCaracter()* es posible que se produzca una excepción y cómo esta se captura en el método *main*:

```
package excepciones;
import java.io.IOException;

public class Excepciones {
    private int caracter;
    public static void main(String[] args) {
        Excepciones ejemplo = new Excepciones();
        try {
            ejemplo.leerCaracter();
            ejemplo.imprimirCaracter();
        } catch (IOException ex) {
            System.out.println(ex.getMessage());
        }
    }
    public void leerCaracter () throws IOException {
        System.out.println("Introduzca carácter a leer:");
        caracter = System.in.read();
    }
    public void imprimirCaracter (){
        System.out.println("Imprimiendo carácter:");
        System.out.println((char)caracter);
    }
}
```

```
run:
Introduzca carácter a leer:
ñ
Imprimiendo carácter:
ñ
```

Actividad propuesta 10.5

¿Qué etiqueta *Javadoc* indica que un método puede propagar una excepción?

10.4.3. Lanzamiento de excepciones

En el ejemplo anterior, el método *System.in.read()* era la instrucción que podría provocar la excepción. Sin embargo, también puede ser el mismo programador el que lance la excepción directamente desde su código. Para ello se hace uso de la instrucción *throw* seguida de la excepción que se desea reportar, siendo esta una excepción existente o bien una desarrollada por el programador. A continuación, puede verse un ejemplo de lanzamiento y captura de excepciones:

```
package excepciones;
public class Excepciones {
    public static void main(String[] args) {
        try {
            Persona persona = new Persona();
            persona.setEdad(-22);
            System.out.println("Edad establecida con éxito");
        }catch (IllegalArgumentException ex1) {
            System.out.println(ex1.getMessage());
        }
    }
}
```

```
run:
La edad debe ser mayor que 0.
```

```
package excepciones;
public class Persona {
    private int edad;
    public int getEdad() {
        return this.edad;
    }
    public void setEdad(int edad) {
        if (edad <= 0)
            throw new IllegalArgumentException("La edad debe ser mayor que 0.");
        this.edad = edad;
    }
}
```


Asimismo, una excepción que haya sido capturada por un bloque *catch* puede volver a ser relanzada para que sea capturada nuevamente por otro bloque *catch* externo:

```
try {
    ... // código susceptible de generar excepción
} catch (java.lang.ClassNotFoundException exc) {
    ... // código que controla la excepción
    throw exc; // relanzando la excepción
}
```

Actividad propuesta 10.6

Describe las diferencias entre *throws* y *throw*. ¿Pueden coexistir en el mismo método?

10.4.4. Creación de clases de excepciones

Otra de las ventajas del uso de excepciones es que se pueden declarar nuevas excepciones para dar cobertura a todos los posibles errores que se quieran gestionar y para representar situaciones excepcionales y específicas de cada programa.

Las excepciones personalizadas deben ser declaradas como subclases de la clase *Exception* (serán *checked*), la clase *RuntimeException* (serán *unchecked*) o cualquier subclase de estas. A continuación, se muestra el mismo ejemplo anterior sobre el que se utiliza una excepción *ad hoc*:

```
package excepciones;
public class Excepciones {
    public static void main(String[] args) {
        try {
            Persona persona = new Persona();
            persona.setEdad(-22);
            System.out.println("Edad establecida con éxito");
        }catch (ExcepcionEdadInvalida ex1) {
            System.out.println(ex1.getMessage());
        }
    }
}
```

```
run:
La edad debe ser mayor que 0.
```

```
package excepciones;
public class Persona {
    private int edad;
    public int getEdad() {
        return this.edad;
    }
    public void setEdad(int edad) {
        if (edad <= 0)
            throw new ExcepcionEdadInvalida("La edad debe ser mayor que 0.");
        this.edad = edad;
    }
}
```

```
package excepciones;
public class ExcepcionEdadInvalida extends IllegalArgumentException {
    ExcepcionEdadInvalida(String mensaje){
        super(mensaje);
    }
}
```

10.5. Recomendaciones de uso de excepciones

El uso de excepciones permite controlar posibles situaciones de error que desemboquen en una finalización errónea del programa, pero su uso no debe ser arbitrario y desordenado, pues de lo contrario pueden dificultar mucho la detección de fallos y el mantenimiento del código. A continuación, se enumeran una serie de recomendaciones de uso de excepciones:

- ✓ *Evitar bloques* catch *vacíos*. Ciertamente estos bloques recogerán las posibles excepciones que se produzcan en el código, pero si no realizan ninguna tarea, los errores pasarán inadvertidos. Se recomienda incluir un mensaje informando del error que haya podido producirse o imprimir una traza completa del error. Ejemplo:

```
try {
   // código susceptible de generar excepciones
} catch (Tipo1 ex){
   System.out.println ("Error: " + ex.getMessage());
   ex.printStackTrace();
}
```

- ✓ *Liberar recursos* mediante el empleo de bloques *finally* o *try-with-resources*.
- ✓ *Evitar lanzar excepciones genéricas* pues aportan poca información. Es mejor lanzar excepciones más específicas que permitan conocer bien el tipo de error y actuar de una forma más precisa. Las excepciones más específicas deben declararse antes que las más genéricas.
- ✓ *Procurar utilizar excepciones existentes* si con ello damos cobertura a nuestro control de excepciones, en lugar de crearlas personalizadas. Se evita así crear nuevas clases de excepciones con un significado que otros programadores desconocerán y se reutiliza código.
- ✓ *Cualquier situación excepcional* que deje a la aplicación en un estado irrecuperable o que no guarde relación directa con el código en el que se ha producido debe ser declarada de tipo *unchecked*.
- ✓ *No abusar de las excepciones y evitar su aparición con códigos bien diseñados*. Un ejemplo fácilmente entendible es sobre las diferentes formas de recorrer un *array*. Es posible hacerlo:
  - Con un bucle *for* que se mueva por todos sus elementos, desde el primero hasta el último (este vendrá dado por la longitud máxima del *array*).
  - Con un bucle *while* desde la primera posición en adelante, hasta que se acceda a una posición inválida y se provoque una excepción que tendrá que ser capturada.

  Como es de suponer, la segunda de las alternativas es poco elegante, más complicada de comprender y hace uso del control de excepciones innecesariamente, lo cual dificultará la optimización de código por parte de la JVM.

Resumen

- Existe una gran variedad de clases de excepciones, aunque también se pueden crear clases que heredan de las excepciones para personalizarlas.

- Las excepciones pueden clasificarse en *checked* (obligatorio controlarlas) y *unchecked*.
- Se pueden capturar diferentes tipos de excepciones para un mismo código encerrado en el bloque *try*. Es importante establecer la captura de las más específicas antes que el de las más genéricas.
- Al lanzarse una excepción automáticamente se sale del bloque de código actual y:
  - Si el bloque tiene asociada una cláusula *catch* para el tipo de la excepción generada, se ejecutará el código del bloque de dicha cláusula y desde ahí continuará el flujo normal de ejecución.
  - Si no, saldrá inmediatamente del bloque o método dentro del cual está el código que generó la excepción y se buscará la cláusula *catch* más adecuada.
  - Este proceso continuará hasta llegar al método *main* de la aplicación, y si finalmente ahí tampoco existe una cláusula *catch* adecuada, la máquina virtual Java finalizará ejecución del programa con un mensaje de error.
- Es posible lanzar excepciones de forma manual mediante la instrucción *throw*.

Ejercicios propuestos

1. Describe la diferencia existente entre excepciones *checked* y *unchecked*.
2. ¿Es posible crear una excepción personalizada con más miembros que los que tiene la clase base? Justifica tu respuesta.
3. Investiga sobre el empleo de *multicatch* y *final rethrow*. Propón un programa de ejemplo de uso en el que se pueda comprobar su funcionamiento.
4. ¿Es correcto el fragmento de código siguiente? Justifica tu respuesta.

```
try {

} finally {

}
```

5. ¿En qué situaciones se podrán capturar excepciones del tipo *IllegalArgumentException* según el siguiente código? Justifica tu respuesta.

```
try {

} catch (Exception ex1) {

} catch (IllegalArgumentException ex2) {

}
```

6. Investiga sobre el método *getCause()* y propón ejemplos en los que sea interesante su uso.
7. ¿Qué relación guardan las interfaces *Closeable* y *AutoCloseable* con el manejo de excepciones?

Supuestos prácticos

1. Modifica el programa *Calculadora* del supuesto práctico 4 que realizaste en el capítulo 5, incluyendo el control de excepciones necesario para evitar salidas incontroladas del programa ante un intento de división por 0 o la entrada de valores no numéricos como operandos. No emplees excepciones personalizadas.
2. Implementa clases excepción necesarias para ser empleadas en el ejercicio anterior y modifica el código para que las utilice.
3. Diseña un programa que haga uso de una función que pregunte por el año de nacimiento de un usuario. Si el año introducido es negativo, la función deberá lanzar una excepción que tendrá que ser capturada en el programa principal.
4. Construye un programa que pida al usuario una cadena de texto. Acto seguido, solicita la posición de un carácter de la cadena de texto e imprímelo. Controla las posibles excepciones producidas al acceder a posiciones inválidas de la cadena.

Actividades de autoevaluación

1. ¿Qué método imprime por la salida de error una traza con las llamadas a los métodos desde los que se ha producido la excepción?
   - ☐ *a)* *getMessage()*
   - ☐ *b)* *getClass()*
   - ☐ *c)* *toString()*
   - ☐ *d)* *printStackTrace()*
2. Una excepción *checked*:
   - ☐ *a)* Generalmente se produce por errores ajenos al código del programa.
   - ☐ *b)* Se corresponde, por ejemplo, con la excepción *RuntimeException*.
   - ☐ *c)* Es obligatorio que sea controlada.
   - ☐ *d)* Todas las opciones anteriores son correctas.
3. Una excepción *unchecked*:
   - ☐ *a)* Se corresponde, por ejemplo, con la excepción *ArrayIndexOutOfBoundsException*.
   - ☐ *b)* Generalmente se produce por errores de programación.
   - ☐ *c)* No debe ser forzosamente capturada.
   - ☐ *d)* Todas las opciones anteriores son correctas.
4. ¿Cuál de los siguientes bloques para control de excepciones no es opcional?
   - ☐ *a)* *finally*.
   - ☐ *b)* *try*.
   - ☐ *c)* *catch*.
   - ☐ *d)* Todas las opciones son incorrectas.

5. Para capturar una excepción:
   - ☐ *a)* Se debe utilizar un bloque *try-catch-finally*.
   - ☐ *b)* Se puede modificar la firma del método añadiendo *throws* y el tipo de excepción a capturar.
   - ☐ *c)* Se puede crear la excepción y lanzarla con la palabra reservada *launch*.
   - ☐ *d)* Se puede crear la excepción y lanzarla con la palabra reservada *throw*.
6. La excepción que se generará al intentar convertir una cadena "tres" en un número entero de tipo primitivo será:
   - ☐ *a)* *java.lang.PrimitiveIntegerTypeException*
   - ☐ *b)* *java.lang.NumberFormatException*
   - ☐ *c)* *java.lang.IllegalArgumentException*
   - ☐ *d)* *java.lang.NullPointerException*
7. ¿Cuál de las siguientes representa una opción a seguir cuando se trata de excepciones?
   - ☐ *a)* Usar bloques *catch* vacíos.
   - ☐ *b)* Lanzar excepciones lo más genéricas posible.
   - ☐ *c)* Crear excepciones para cada tipo de error por controlar.
   - ☐ *d)* Todas las opciones anteriores son incorrectas.
8. Llevar un buen control de las excepciones permite:
   - ☐ *a)* Agrupar los diferentes tipos de errores y proponer códigos alternativos para tratarlos.
   - ☐ *b)* Propagar los errores sobre la pila de llamadas.
   - ☐ *c)* Separar el código destinado a controlar errores del resto del código normal y diseñado para un fin concreto.
   - ☐ *d)* Todas las opciones anteriores son correctas.
9. Una excepción personalizada:
   - ☐ *a)* Puede ser declarada como subclase de *Exception*.
   - ☐ *b)* Puede ser declarada como subclase de *RuntimeException*.
   - ☐ *c)* Las opciones *a)* y *b)* son correctas.
   - ☐ *d)* Todas las opciones anteriores son incorrectas.
10. ¿Cuál de las siguientes afirmaciones es falsa?
    - ☐ *a)* Las características *multicatch* y *try-with-resources* están disponibles desde Java SE 7.
    - ☐ *b)* El uso de *try-with-resources* no es compatible con el de bloques *finally*.
    - ☐ *c)* Si es necesario capturar varias excepciones, primero se capturan las más específicas.
    - ☐ *d)* Es importante liberar los recursos empleados mediante el empleo de bloques *finally* o *try-with-resources*.

SOLUCIONES:

1.	a	b	c	**d**	5.	**a**	b	c	d						
2.	a	b	**c**	d	6.	a	**b**	c	d	9.	a	b	**c**	d	
3.	a	b	c	**d**	7.	a	b	c	**d**	10.	a	**b**	c	d	
4.	a	**b**	c	d	8.	a	b	c	**d**						

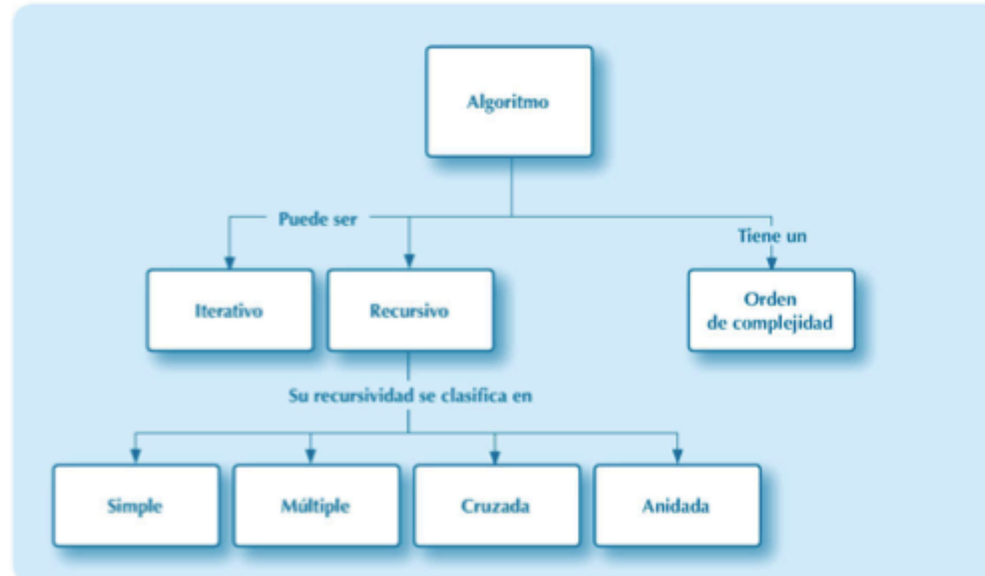
11

Recursividad y complejidad algorítmica

Objetivos

- ✓ Conocer el concepto de recursividad.
- ✓ Distinguir entre los diferentes tipos de recursividad.
- ✓ Aprender a desarrollar soluciones recursivas e iterativas equivalentes.
- ✓ Saber valorar el coste de utilizar una solución recursiva frente a una iterativa equivalente.
- ✓ Dominar las ventajas e inconvenientes inherentes al uso de la recursividad.
- ✓ Entender el concepto de complejidad algorítmica.
- ✓ Aprender a calcular el orden de complejidad de algoritmos iterativos y recursivos.

Mapa conceptual



Glosario

Búsqueda dicotómica. También conocida como búsqueda binaria, es un algoritmo de búsqueda sobre colecciones ordenadas que busca un elemento dividiendo la colección en 2 partes e identificando en cuál de ellas se encuentra. Posteriormente vuelve a dividir la parte que contiene el elemento buscado en 2 mitades iguales y busca de nuevo. Se repetirá el proceso hasta encontrar el elemento.

Caso base. Se denomina caso base de un algoritmo recursivo a cada uno de los problemas de pequeño tamaño en los que se descompone un problema para ser resuelto y que no genera nuevas llamadas recursivas.

Factorial. El factorial de un número entero positivo *n* es el producto de todos los números positivos desde n hasta 1. Se representa como *n!*

Fractal. Figura geométrica cuya estructura básica se repite a diferentes escalas.

Matrioshka. También conocida como muñeca rusa, es un conjunto impar de muñecas huecas en el que cada muñeca aloja en su interior una nueva muñeca de aspecto similar y menor escala.

Merge sort. Algoritmo recursivo de ordenación de elementos de una colección en la que se va dividiendo recursivamente la lista desordenada en dos sublistas (de aproximadamente el mismo tamaño). Una vez se tienen los elementos separados, se van mezclando las dos sublistas en una única lista.

Notación *Big-O*. También conocida como notación asintótica o notación Landau, es la forma de representar el orden de complejidad de un algoritmo.

Orden de complejidad. Es la variable que permite medir el rendimiento y eficiencia de un determinado algoritmo.

***Quick sort*.** Es un algoritmo recursivo de ordenación rápida que se basa en elegir un elemento pivote de entre todos los elementos de la colección y recolocar el resto de elementos ubicados en cada lado del pivote a un lado u otro de este según su valor. Este proceso se repetirá con los elementos situados a un lado y otro del pivote, eligiendo los respectivos nuevos pivotes de estos subconjuntos y repitiendo el proceso mientras haya elementos en ambos lados.

Recursividad. También conocida como recursión o recurrencia, es la técnica de programación que emplea llamadas recurrentes de una función sobre sí misma para resolver un determinado problema.

Segmentación. Técnica de solapamiento de instrucciones en un procesador mediante la división de su ejecución en diferentes segmentos.

11.1. Introducción

Llegados a este punto seguramente ya se hayan desarrollado varios programas en los que siempre se ha llevado a cabo una lógica de diseño basada en algoritmos iterativos. Sin embargo, existe una alternativa a la hora de resolver determinados problemas conocida como *recursividad*. Este concepto, que es uno de los que más trabajo cuesta afianzar a un programador una vez que ha adquirido las destrezas mínimas en el desarrollo de programas, será el que se someterá a estudio en el presente capítulo.

11.2. Concepto de *recursividad*

Habitualmente se utiliza la *iteración* para intentar repetir un conjunto de instrucciones sin tener que escribir el código una y otra vez mediante las típicas estructuras de control iterativas (*for*, *while* y derivados). Sin embargo, muchos de esos programas pueden ser resueltos con *recursividad* de forma más sencilla.

Aunque el concepto *recursividad*, *a priori*, asusta (pues parece complejo y abstracto), en realidad es bastante simple. La técnica de la *recursividad* consiste en definir una función, o método si esto se extrapola a programación orientada a objetos, en función de sí misma. Dicho de una forma menos teórica, la recursividad es el proceso mediante el cual una función se llama a sí misma una y otra vez hasta satisfacer una condición de finalización. Se puede decir que la recursividad descompone el problema en partes cada vez más pequeñas, de forma similar a como se haría el desmontaje de una *matrioshka*.

La recursividad tiene, por tanto, una relación muy directa con la *programación modular* y con el lema "divide y vencerás" que propone la descomposición de problemas de tamaño *N* en subproblemas de menor tamaño que, a su vez, serán descompuestos sucesivamente en otros, repitiendo el proceso hasta que el resultado de dichas descomposiciones permita obtener problemas fáciles de resolver.

Investiga

Busca información sobre el concepto *fractal*. ¿Crees que guarda alguna relación con el concepto *recursividad*?

11.3. Utilización de la recursividad

En líneas generales, para hacer uso de la recursividad se deben realizar invocaciones de una función o método desde sí misma. Pero para utilizar la recursividad correctamente es importante conocer cuáles son los *casos base* que determinarán la finalización de esa llamada. A partir de ellos será posible ir construyendo la solución. En caso contrario, lo que se estará construyendo serán bucles infinitos que únicamente finalizarán cuando se hayan consumido todos los recursos asociados al programa. El trágico resultado de esos bucles puede verse en el siguiente ejemplo, donde finalmente se produce un desbordamiento de pila y se genera una excepción *java.lang.StackOverflowError* que anuncia que la máquina virtual de Java se ha quedado sin espacio suficiente para incluir una nueva llamada:

```
package recursividad;
public class Recursividad {
    void metodoA() {
        metodoA();
    }
    public static void main(String[] ar) {
        Recursividad recursivo=new Recursividad();
        recursivo.metodoA();
    }
}
```

```
run:
Exception in thread "main" java.lang.StackOverflowError
        at recursividad.Recursividad.metodoA(Recursividad.java:4)
        at recursividad.Recursividad.metodoA(Recursividad.java:4)
        at recursividad.Recursividad.metodoA(Recursividad.java:4)
```

Antes de mostrar ejemplos correctos de programas recursivos, hay que entender cuál es la estructura que tendrá un programa recursivo y la diferencia frente a un programa iterativo. En un programa iterativo, un método irá invocando a diferentes métodos, cooperando de esta manera entre sí para obtener la solución; en un programa recursivo, un método invocará siempre a sí mismo mientras que se cumpla una determinada condición.

Figura 11.1
Llamadas iterativas versus llamadas recursivas.

11.4. Tipos de recursividad

Cuando se habla de recursividad es importante distinguir entre los diferentes tipos que pueden encontrarse:

1. *Recursividad simple*: en este tipo de recursividad únicamente aparece una llamada recursiva. Es fácil cambiar este tipo de solución por otra iterativa equivalente. Un ejemplo de este tipo de recursividad es la implementación del siguiente programa que realiza la *cuenta atrás* desde 10 (caso base es $n<=0$):

SOLUCIÓN RECURSIVA

```
public class Ejemplo {
    void cuentaAtras(int n) {
        if (n>0){
            System.out.println(n);
            cuentaAtras(--n);
        }else
            System.out.println("Final !");
    }
    public static void main(String[] ar) {
        Ejemplo ejemplo=new Ejemplo();
        ejemplo.cuentaAtras(10);
    }
}
```

SOLUCIÓN ITERATIVA

```
public class Ejemplo {
    void cuentaAtras(int n) {
        while (n>0){
            System.out.println(n--);
        }
        System.out.println("Final !");
    }
    public static void main(String[] ar) {
        Ejemplo ejemplo=new Ejemplo();
        ejemplo.cuentaAtras(10);
    }
}
```

```
run:
10
9
8
7
6
5
4
3
2
1
Final !
```

Otro ejemplo resuelto típicamente de forma recursiva es el del cálculo del *factorial* de un número. El número *factorial* de *n* se calcula como el producto de todos los números enteros positivos desde *n* hasta *1*:

SOLUCIÓN RECURSIVA

```
public class Ejemplo {
    int factorial(int n) {
        if (n>0){
            int resultado = n * factorial (n-1);
            return resultado;
        }else
            return 1;
    }
    public static void main(String[] ar) {
        Ejemplo ejemplo=new Ejemplo();
        System.out.println("Factorial de "+6);
        System.out.println(ejemplo.factorial(6));
    }
}
```

SOLUCIÓN ITERATIVA

```
public class Ejemplo {
    int factorial(int n) {
        int resultado = 1;
        for (int i=n;i>=1;i--){
            resultado *= i;
        }
        return resultado;
    }
    public static void main(String[] ar) {
        Ejemplo ejemplo=new Ejemplo();
        System.out.println("Factorial de "+6);
        System.out.println(ejemplo.factorial(6));
    }
}
```

```
run:
Factorial de 6
720
```

2. *Recursividad múltiple*: se da cuando en el cuerpo del método o función hay más de una llamada al mismo método o función. Este tipo de soluciones son más complejas de transformar en iterativas. Un ejemplo de este tipo de recursividad es el siguiente, que se emplea para buscar el elemento de mayor valor dentro de un *array* estático:


```
run:
El número máximo entre:
[5, 22, 13, 43, 77, 17, 19, 33, 10]
es: 77
```

```
package recursividad;
import java.util.Arrays;
public class Ejemplo {
    int Maximo(int[] numeros, int limiteInferior, int limiteSuperior) {
        if (limiteInferior==limiteSuperior)
            return numeros[limiteInferior];
        else {
            int posicionMedia = (limiteInferior + limiteSuperior)/2;
            int maximoInferiores = Maximo (numeros, limiteInferior, posicionMedia);
            int maximoSuperiores = Maximo (numeros, posicionMedia+1, limiteSuperior);
            return (Math.max(maximoInferiores, maximoSuperiores));
        }
    }
    public static void main(String[] ar) {
        Ejemplo ejemplo=new Ejemplo();
        int[] numeros = {5, 22, 13, 43, 77, 17, 19, 33, 10};
        System.out.println("El número máximo entre:");
        System.out.println(Arrays.toString(numeros));
        System.out.println("es: "+ejemplo.Maximo(numeros,0,numeros.length-1));
    }
}
```

3. *Recursividad cruzada o indirecta:* se presenta en aquellos escenarios en los que un método o función provoca una llamada a sí misma de forma indirecta a través de otras funciones (*A* invoca a *B*, y a continuación *B* invoca a *A*). Un ejemplo típico de este tipo de recursividad es el siguiente, en el que se comprueba si un número es par o impar:

```
public class Ejemplo{
    public static boolean par(int n){
        if(n==0) return true;
        else return impar(n-1);
    }
    public static boolean impar(int n){
        if(n==0) return false;
        else return par(n-1);
    }
    public static void main(String [] args){
        int n = 8;
        int m = 11;
        System.out.println(n + " es par --> " + par(n));
        System.out.println(m + " es par --> " + par(m));
    }
}
```

```
run:
8 es par --> true
11 es par --> false
```

4. *Recursividad anidada*: se da cuando en los argumentos de la llamada recursiva hay llamadas a la misma función o método. Un ejemplo típico de este tipo de recursividad es la función de *Ackermann*:

```
public static long ackermann(long m, long n){
    if (m == 0)
        return (n + 1);
    else if (n == 0)
        return (ackermann(m - 1, 1));
    else
        return (ackermann(m - 1, ackermann(m, n - 1)));
}
```

Como ha podido comprobarse, la recursividad no siempre se utiliza con métodos que devuelven un valor determinado tras su invocación, sino que en algunas ocasiones simplemente van realizando tareas sobre un conjunto de datos.

Para saber más

La función de *Ackermann* es utilizada para probar los niveles de optimización de un compilador.

Recurso web

En el siguiente enlace de Wikipedia, podrás encontrar más información sobre el concepto *factorial* empleado en el ejemplo de *recursividad simple*.

Actividades propuestas

11.1. Identifica los casos base de los ejemplos recursivos mostrados en el capítulo.

11.2. Implementa un programa recursivo que, dado un número entero positivo en base decimal, devuelva su equivalente en representación binaria.

11.5. Ventajas e inconvenientes

Una vez explicado el concepto *recursividad* y haberse mostrado los diferentes tipos existentes, se puede realizar una valoración sobre las ventajas e inconvenientes de su uso:

- Un algoritmo recursivo consume más memoria y es más lento que el algoritmo iterativo que resuelve el mismo problema. Esto se debe a que, por cada nueva llamada, hay que crear y almacenar un nuevo contexto. Como consecuencia, algunos problemas que podrían resolverse muy fácilmente con algoritmos recursivos son, físicamente hablando, imposibles de resolver por esta vía, pues agotan los recursos del computador antes de encontrar la solución.
- Un algoritmo recursivo ocupa menos espacio que un algoritmo iterativo que resuelve el mismo problema y, generalmente, se presenta como una solución más elegante.
- La recursividad se plantea como un reto complejo para diseñadores de arquitecturas hardware y compiladores, pues dificulta en gran medida la segmentación de instrucciones y su optimización.
- Un algoritmo recursivo aplica fielmente la técnica "divide y vencerás", partiendo para ello de la búsqueda de los casos base. Sin embargo, un algoritmo iterativo consigue llegar a la solución mediante el empleo de bucles e instrucciones secuenciales.

11.6. Complejidad algorítmica

La complejidad algorítmica no tiene una relación directa con la recursividad, pues se trata de un concepto que viene a representar la cantidad de recursos que un algoritmo (ya sea *iterativo* o *recursivo*) necesitará para resolver un problema determinado en el peor de los casos. Sin embargo, ahora que se ha explicado la recursividad es cuando se puede comprender mejor este concepto que permitirá determinar cuán buena es una solución frente a otra.

En realidad, puede existir un gran número de parámetros a tener en cuenta, pero generalmente el estudio se centra en el *tiempo de ejecución* y el *espacio de memoria consumida*. La complejidad se mide utilizando la *notación Big-O* que representa la eficiencia del algoritmo y que permite conocer cómo se comportará en función del número de argumentos que reciba. Las *órdenes de complejidad* existentes se exponen en el cuadro 11.1.

Cuadro 11.1
Órdenes de complejidad algorítmica

Orden	Nombre	Descripción
O(1)	constante	La complejidad no variará en función del tamaño del dato.
O(log n)	logarítmica	Suele corresponder a algoritmos que acotan el problema para abordarlo. Se puede encontrar en bucles en los que la evolución de la variable de control no es lineal o en algoritmos recursivos de *búsqueda dicotómica*.
O(n)	lineal	La complejidad es directamente proporcional al tamaño de los datos. Se puede encontrar en bucles que realizan *N* iteraciones (*while* o *for*), algoritmos de búsqueda lineal, cálculo de factorial…
O(n log n)	cuasi lineal	Se encuentra en algoritmos que aplican el lema "divide y vencerás". Resuelven el problema dividiéndolo en problemas más sencillos que resuelve por separado para posteriormente formar la solución. Algunos ejemplos son los algoritmos de ordenación rápida (*quick sort*) o el de ordenación por mezcla (*merge sort*).

$O(n^2)$	cuadrática	Se presenta cuando hay que evaluar todos los emparejamientos de elementos de un conjunto: • *Cuadrática* → bucles anidados de 2 niveles (manejo de matrices). • *Cúbica* → bucles anidados de 3 niveles. • *Polinomial* → bucles anidados de *a* niveles (*a*>2).
$O(n^3)$	cúbica	
$O(n^a)$	polinomial	
$O(a^n)$	exponencial	En este tipo de orden *a*>*1*. La complejidad crece enormemente a medida que van creciendo los datos de entrada. Un ejemplo típico son los algoritmos que resuelven problemas por fuerza bruta, los que calculan la secuencia de Fibonacci o el algoritmo de las torres de Hanoi.
$O(n!)$	factorial	Es la menos deseable de todas y generalmente se asocia con problemas intratables o resolubles mediante fuerza bruta.

Los órdenes de complejidad responden a la jerarquía siguiente en la que establece que, a menor orden, menor cantidad de recursos necesarios para resolver el problema:

$$O(1) < O(\text{Log } n) < O(n) < O(n \text{ Log } n) < O(n^a) < O(a^n) < O(n!)$$

Sin entrar en muchos detalles ni complejas operaciones matemáticas, para facilitar el cálculo de la complejidad se enumeran las siguientes reglas:

- ✓ Se considerará que la complejidad será lineal para todas aquellas operaciones de entrada-salida, sentencias de asignación, evaluación de expresiones aritméticas sobre datos de tamaño constante, comparaciones de datos simples, accesos a elementos de un *array*…
- ✓ El orden de complejidad de un conjunto de instrucciones será la suma de los órdenes de complejidad de los elementos individuales.
- ✓ En caso de sentencias condicionales (*if-else*), la complejidad del conjunto será siempre la mayor de las complejidades de todas las alternativas.
- ✓ Si un bucle contiene una instrucción de complejidad *O(1)* y se ejecuta un número fijo de veces independiente del tamaño de los datos, la complejidad se simplifica a la que tenga la instrucción contenida, que en este caso sería *O(1)*.
- ✓ Si un bucle contiene una instrucción de complejidad *O(1)* y se ejecuta un número de veces dependiente del tamaño de los datos (*N*), la complejidad se considerará lineal; en este caso *O(n)*.
- ✓ La complejidad de las llamadas a procedimientos, funciones o métodos, se supone nula y únicamente vendrá dada por la complejidad de las instrucciones en él contenidas.
- ✓ Si el programa es recursivo, hay que distinguir entre la complejidad del caso base y de los recurrentes.

Sabías que…

El juego de Las Torres de Hanoi es un rompecabezas que propone mover los discos apilados sobre un poste a) hasta un poste c) para que queden apilados exactamente igual con el menor número de movimientos posible. Se puede utilizar un poste b) a modo de elemento auxiliar. Como normas base no se puede mover más de un disco a la vez, el apilamiento solo está permitido sobre el disco de tamaño inmediatamente inferior y los discos siempre deben estar en alguno de los postes:

Actividades propuestas

11.3. Investiga sobre el algoritmo de *búsqueda dicotómica* y realiza su implementación sobre un *array* estático de 20 elementos ordenados.

11.4. Busca información sobre el algoritmo de ordenación *merge sort* y realiza su implementación sobre un *array* estático de 20 elementos desordenados.

11.5. Indaga sobre el algoritmo de ordenación *quick sort* y realiza su implementación sobre un *array* estático de 20 elementos desordenados.

Resumen

- En la vida cotidiana y en la naturaleza, existen estructuras cuya forma y composición guardan relación con el concepto *recursividad*.
- La recursividad es otra forma normalmente más elegante, compacta y sencilla, de resolver algunos problemas.
- El concepto *recursividad* rompe con la idea tradicional de que un método *A* siempre invoca a un método *B* distinto y encaja perfectamente con el lema "divide y vencerás".

- Existen varios tipos de recursividad: simple, múltiple, anidada y cruzada.
- Generalmente la solución recursiva es menos eficiente en términos de tiempo y consumo de memoria, ya que debe crear espacios nuevos en memoria en cada llamada para almacenar la información de las variables.
- Algunos problemas son teóricamente resolubles mediante técnicas recursivas, pero físicamente imposibles de resolver pues agotarían los recursos del ordenador antes de obtener solución.
- La complejidad algorítmica permite conocer cuán eficiente o costoso es un algoritmo.

Ejercicios propuestos

1. ¿Es posible crear un programa recursivo sin caso base? Justifica tu respuesta.
2. Si un método o función recursivo no declara nuevas variables en su cuerpo pero no existe caso base, ¿es posible que se consuman todos los recursos y se aborte el programa? Justifica tu respuesta.
3. ¿Se pueden encontrar varios tipos de recursividad en el mismo algoritmo? Justifica tu respuesta.
4. Evalúa el siguiente fragmento de código e indica qué tipo de recursividad se está empleando y su complejidad:

```
public static int funcion(int n){
    if (n==1 || n==2) return 1;
    return funcion(n-1) + funcion(n-2);
}
```

5. Si tuvieses que elaborar un programa destinado a generar contraseñas hasta conseguir la correcta (ejemplo típico de ataque por fuerza bruta), ¿emplearías una solución iterativa o recursiva? Justifica tu respuesta.

Supuestos prácticos

1. Realiza la implementación de un programa recursivo que imprima por pantalla el valor de una variable entera que irá incrementando en una unidad hasta que la pila se desborde, e indica el número de llamadas que ha sido posible realizar de forma recursiva.

2. Modifica el programa anterior para que capture la excepción producida al agotarse el espacio disponible en la pila de llamadas y salga del programa ordenadamente.
3. Implementa un programa recursivo que lea una cadena de caracteres por teclado y que, a continuación, vaya eliminando de la cadena el último carácter. Esta operación deberá repetirse hasta conseguir tener un solo elemento en la cadena. En cada llamada al método deberá imprimirse el contenido actual de la cadena.
4. Investiga sobre el concepto matemático *sucesión de Fibonacci* y construye un programa recursivo y otro iterativo que imprima por pantalla la sucesión de los *n* primeros números de la sucesión.
5. Implementa un programa recursivo que realice la división de un número *n* por otro *m* mediante el empleo de sucesivas restas.
6. Programa un algoritmo que primero construya un *array* de 15 números aleatorios comprendidos entre 0 y 9 para posteriormente multiplicar todos los elementos contenidos mediante un método recursivo.

Actividades de autoevaluación

1. En un programa recursivo:
   - ☐ *a)* Siempre debe existir un único caso base.
   - ☐ *b)* Pueden existir varios casos base.
   - ☐ *c)* No tiene que existir ningún caso base.
   - ☐ *d)* Todas las opciones son incorrectas.

2. Un programa recursivo:
   - ☐ *a)* Es más rápido que su equivalente iterativo.
   - ☐ *b)* Consume más recursos que su equivalente iterativo.
   - ☐ *c)* Las opciones *a)* y *b)* son correctas.
   - ☐ *d)* Todas las opciones anteriores son incorrectas.

3. ¿Qué tipo de excepción podemos obtener en nuestro programa si no controlamos correctamente la recursividad?
   - ☐ *a)* *java.lang.OverflowErrorStack*
   - ☐ *b)* *java.lang.ErrorStackOverflow*
   - ☐ *c)* *java.lang.StackOverflowError*
   - ☐ *d)* *java.lang.StackErrorOverflow*

4. Cuando se habla de recursividad anidada:
   - ☐ *a)* En el cuerpo de la función o método hay al menos una invocación recursiva.
   - ☐ *b)* En los argumentos de la llamada recursiva hay llamadas a la misma función o método.
   - ☐ *c)* Las opciones *a)* y *b)* son correctas.
   - ☐ *d)* Todas las opciones anteriores son incorrectas.

5. Si *metodo1* invoca a *metodo2*, *metodo2* invoca a *metodo3* y *metodo3* invoca a *metodo1*, ¿se puede hablar de recursividad cruzada?
   - ☐ *a)* Sí, es un claro ejemplo de recursividad cruzada.
   - ☐ *b)* No, puesto que los métodos son independientes.
   - ☐ *c)* No, la recursividad cruzada solo se da entre 2 métodos o funciones.
   - ☐ *d)* Todas las opciones son incorrectas.

6. En cada llamada recursiva:
   - ☐ *a)* Se crea un nuevo contexto.
   - ☐ *b)* Se ocupan recursos nuevos.
   - ☐ *c)* Se puede volver a hacer una nueva invocación a la misma función o método.
   - ☐ *d)* Todas las opciones son correctas.

7. Suponga una estructura alternativa de control *if-else*. Si el bloque de instrucciones dentro del bloque *if* tiene orden de complejidad O(1) y el bloque de instrucciones del bloque *else* correspondiente tiene orden de complejidad O(n):
   - ☐ *a)* El orden de complejidad de la estructura *if-else* es O(1).
   - ☐ *b)* El orden de complejidad de la estructura *if-else* es O(n).
   - ☐ *c)* El orden de complejidad de la estructura *if-else* es O(1) + O(n).
   - ☐ *d)* El orden de complejidad de la estructura *if-else* es O(1) * O(n).

8. Selecciona el orden jerárquico correcto de los siguientes órdenes de complejidad:
   - ☐ *a)* $O(1) < O(\log n) < O(n) < O(n \log n) < O(2^n) < O(n^2) < O(n^3)$
   - ☐ *b)* $O(1) < O(n) < O(\log n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$
   - ☐ *c)* $O(1) < O(\log n) < O(n \log n) < O(n) < O(n^2) < O(n^3) < O(2^n)$
   - ☐ *d)* $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$

9. El caso base de un algoritmo recursivo:
   - ☐ *a)* Siempre es único, no pudiendo existir más de uno.
   - ☐ *b)* Debe evaluarse después de hacer la llamada recursiva.
   - ☐ *c)* Representa el punto de finalización de una llamada recursiva.
   - ☐ *d)* Tiene una complejidad asociada *O(n)*.

10. Si un bucle contiene una instrucción de complejidad *O(1)* y se ejecuta un número *k* de veces independiente del tamaño de los datos:
    - ☐ *a)* Tendrá una complejidad *O(1)*.
    - ☐ *b)* Tendrá una complejidad *O(n)*.
    - ☐ *c)* Tendrá una complejidad *O(k)*.
    - ☐ *d)* Tendrá una complejidad *O(n log n)*.

SOLUCIONES:

1. ☐ a ■ b ☐ c ☐ d	5. ■ a ☐ b ☐ c ☐ d		
2. ☐ a ■ b ☐ c ☐ d	6. ☐ a ☐ b ☐ c ■ d	9. ☐ a ☐ b ■ c ☐ d	
3. ☐ a ☐ b ■ c ☐ d	7. ☐ a ■ b ☐ c ☐ d	10. ■ a ☐ b ☐ c ☐ d	
4. ☐ a ☐ b ■ c ☐ d	8. ☐ a ☐ b ☐ c ■ d		

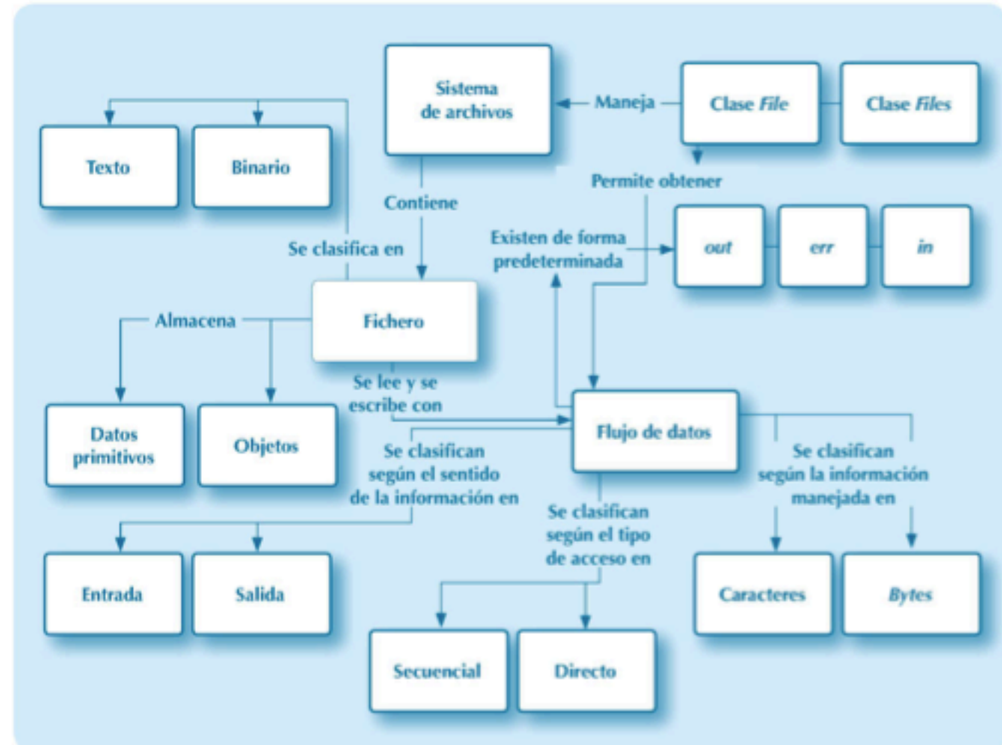
12

Lectura y escritura de información

Objetivos

- ✓ Conocer el concepto de *flujo de datos*.
- ✓ Distinguir los diferentes flujos de datos.
- ✓ Saber aplicar el tipo de flujo de datos más adecuado a cada situación y tipo de información.
- ✓ Dominar los pasos a seguir para utilizar un flujo de datos.
- ✓ Aprender a hacer persistente la información manejada por un programa.
- ✓ Estar familiarizado con los distintos tipos de ficheros, así como las clases y métodos que permiten manejarlos.
- ✓ Comprender el concepto *serialización* y aprender a manejarlo.
- ✓ Utilizar las herramientas proporcionadas por el lenguaje Java para el manejo de sistemas de ficheros.

Mapa conceptual



Glosario

Archivo disperso. Archivo en el que gran parte de los datos son ceros y el resto se encuentra muy repartido entre ellos.

Buffer. Representa un espacio en memoria en el que se van almacenando datos de forma temporal para ser leídos por otro programa, proceso o hilo de ejecución.

Deserialización. Transformación de una secuencia de *bytes* a un objeto.

Flujo. Secuencia ordenada de datos que se transmite desde una fuente hasta un destino. En Java se denomina *Stream*.

Lorem ipsum. Texto formateado que permite su inclusión en documentos a modo de relleno.

Multihilo. Se dice que un programa es *multihilo, multithread* o *multihebra* cuando hay partes de su código que son ejecutadas de forma simultánea. No se debe confundir con el término *multiproceso,* que hace referencia a procesos (programas) independientes que se ejecutan simultáneamente.

Path. Es la ruta o camino que permite encontrar un determinado recurso dentro de un sistema de archivos.

Serialización. Transformación de objetos a una secuencia de *bytes* que puede ser almacenada en un fichero binario.

Tubería. También conocido como *Pipe,* permite transferir el contenido (flujo) que hay en un extremo a otro. Este concepto también es muy utilizado en sistemas operativos para redirigir la salida de un comando (resultados) como argumentos para la llamada de otro.

Von Neumann. John von Neumann fue un importante y destacado matemático que realizó contribuciones elementales en el campo de la computación. Los ordenadores modernos basan su diseño y funcionamiento en su arquitectura.

12.1. Introducción

Uno de los aspectos más importantes de cualquier programa es la capacidad de diálogo e intercambio de información con el usuario. En el capítulo 5 (Estructuras de control), se trabajó ya sobre este tema, exponiéndose los principales flujos de información existentes que permiten dotar de interactividad a un programa Java. En este capítulo se tomará de base todo lo expuesto anteriormente para dar un paso más en el manejo de los flujos de comunicación y en el almacenamiento de información.

12.2. Flujos de comunicación

Existe una gran variedad de lenguajes de programación con los que desarrollar programas y existen varias técnicas para el desarrollo de estos, pero lo que todos ellos tienen en común es que, en mayor o menor medida, siempre interactúan con algún elemento externo. De hecho, incluso el programa más sencillo que se pueda imaginar y que siempre es explicado en todo curso de programación, el "Hola Mundo", también lo hace mostrando un simple mensaje por pantalla. Es por ello por lo que se hace indispensable para cualquier lenguaje de programación ofrecer los mecanismos esenciales de entrada y salida de información.

Esta entrada y salida de información se realiza mediante el empleo de flujos de datos que operan a modo de interfaz de comunicación. Un flujo, por tanto, es una secuencia ordenada de datos que se transmite desde una fuente hasta un destino y que en Java se denomina *Stream*. El origen o el destino pueden ser un fichero, un *String* o un dispositivo (teclado, pantalla, altavoz, red…).

12.2.1. Tipos de flujo

En Java existen varios tipos de flujos de datos y estos pueden clasificarse según varios aspectos como pueden ser el tipo de información que manejan, la dirección del flujo de datos o la forma en la que se realiza el acceso:

1. Según la *dirección del flujo de datos* se puede hacer una primera clasificación que distingue entre *flujos de entrada*, *flujos de salida* y *flujos de entrada-salida*.
2. Según la forma en la que se realiza el *acceso*, se puede distinguir entre flujos de *acceso secuencial* o de *acceso directo*.
3. También se puede hacer una clasificación atendiendo al *tipo de información* que manejan. Se pueden encontrar:

   a) *Flujos de caracteres* o *character streams*: empleados para el envío y recepción de caracteres codificados en Unicode. Los datos transferidos son entendibles por humanos. Un ejemplo típico de uso de este tipo de flujos es para la lectura de datos desde teclado, la escritura de datos por pantalla o la lectura-escritura de información hacia un fichero de texto plano (*.txt*).

   b) *Flujos de bytes* o *byte streams:* que permiten la lectura y escritura de *bytes*. Están diseñados para el envío de datos binarios. Los datos transferidos no serán legibles por personas sino por otros programas. Un ejemplo típico de uso de este tipo de flujos es la lectura-escritura de información hacia un fichero binario.

Recurso web

Los datos codificados en Unicode se almacenan siguiendo una codificación diseñada para el intercambio, procesamiento y presentación de textos escritos en varias lenguas. En la URL https://unicode.org/ podrás acceder a toda la información relativa a este estándar de codificación.

12.2.2. Flujos predeterminados

Existen algunos flujos estándar que están disponibles para ser usados sin necesidad de crearlos mientras que el programa esté en ejecución (*System.out*, *System.err* y *System.in*). Estos ya fueron utilizados en el apartado 5.2.2 para permitir la entrada de datos desde teclado y la salida de información por pantalla, por lo que no se añadirán nuevos ejemplos.

Actividad propuesta 12.1

Repasa las definiciones y ejemplos de flujos predeterminados que fueron expuestos en el apartado 5.2.2 y prepara una tabla resumen en la que se recojan sus definiciones y métodos principales.

12.2.3. Utilización de flujos

Antes de entrar en detalle en las clases y métodos que se utilizan para el manejo de flujos de información, es importante conocer cómo deben utilizarse. Todos los flujos, *a priori*, funcionan igual independientemente de cuál sea la fuente de los datos, por lo que para llevar a cabo la lectura o escritura de datos se deberán seguir los siguientes pasos:

1. Crear el *Stream* para abrir el flujo a una fuente de datos.
2. Leer-escribir (dependiendo del tipo de operación) mientras haya datos disponibles.
3. Cerrar el *Stream*.

En el caso de que se empleen flujos estándar, el propio entorno será el que instancie el *Stream* cuando se ejecute el programa y lo cerrará una vez finalice. Por este motivo con estos flujos no es necesario realizar los pasos 1 y 3.

12.2.4. Clases relativas a flujos

Existe una serie de clases que permiten abstraer al programador de los detalles del dispositivo físico con el que se desea intercambiar información, representando a dicho dispositivo como el origen o destino de un flujo de datos. Todas estas clases se encuentran ubicadas en el paquete *java.io*, donde *io* es la abreviatura de *input-output*.

En este apartado serán descritas las clases relacionadas con la entrada y salida de información, haciéndose una clasificación conforme al tipo de información que manejan.

A) Clases para manejo de flujos de bytes

Las clases que representen un flujo de entrada de *bytes* heredarán de las clases abstractas *InputStream* y *OutputStream*. Estas implementan, al menos, los métodos descritos en los cuadros 12.1 y 12.2.

CUADRO 12.1
Métodos de *InputStream*

`int read()`	Lee el siguiente *byte* desde el *InputStream*. Si retorna -1, indicará que no se pueden leer más *bytes*.
`int available()`	Devuelve el número de *bytes* que se pueden leer del *InputStream* sin un bloqueo.
`void close()`	Cierra el *InputStream*.

CUADRO 12.2
Métodos de *OutputStream*

`void write(int b)`	Escribe un *byte* en el *OutputStream*.
`void write(byte[] a)`	Escribe todos los *bytes* del *array* a en el *OutputStream*.
`void flush()`	Vacía el flujo de salida actual del *OutputStream*.
`void close()`	Cierra el *OutputStream* y escribe los contenidos que haya en ese momento.

A continuación, se enumeran y describen las clases principales para el manejo de flujos. Además, se incluye un ejemplo de uso para cada una de ellas:

1. *FileOutputStream:* permite escribir *bytes* hacia un fichero binario.

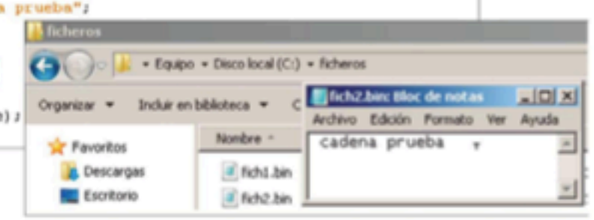
```
byte CR=13; //Salto de linea
byte LF=10; //retorno de carro
try{
    FileOutputStream fos = new FileOutputStream("C:\\ficheros\\fich1.bin");
    fos.write((byte)'a');fos.write((byte)'e');fos.write((byte)'i');
    fos.write(CR);fos.write(LF);
    fos.write((byte)'o');fos.write((byte)'u');
    fos.write(CR);fos.write(LF);
    fos.close();
}catch(IOException ioe){
    System.out.println(ioe);
}
```

2. *BufferedOutputStream:* posibilita escribir información de otro *OutputStream* utilizando un *buffer* interno que mejora el rendimiento.

```
byte CR=13; //Salto de linea
byte LF=10; //retorno de carro
try{
    FileOutputStream fos = new FileOutputStream("C:\\ficheros\\fich1.bin");
    BufferedOutputStream bos=new BufferedOutputStream(fos);
    bos.write((byte)'a');bos.write((byte)'e');bos.write((byte)'i');
    bos.write(CR);bos.write(LF);
    bos.write((byte)'o');bos.write((byte)'u');
    bos.write(CR);bos.write(LF);
    bos.close();
    fos.close();
}catch(IOException ioe){
    System.out.println(ioe);
}
```

3. *DataOutputStream:* define los métodos *writeBoolean*, *writeByte*, *writeInt*…, que permiten escribir datos de tipo primitivo desde un *OutputStream*. Como puede verse en el ejemplo, no todo el contenido del fichero es legible ya que es un fichero binario.

```
try{
    FileOutputStream fos = new FileOutputStream("C:\\ficheros\\fich2.bin");
    DataOutputStream dos = new DataOutputStream(fos);
    String texto = "cadena prueba";
    dos.writeUTF(texto);
    int numero = 22;
    dos.writeInt(numero);
}catch(IOException ioe){
    System.out.println(ioe);
}
```



4. *ByteArrayOutputStream*: posibilita usar un *array* de *bytes* como un *OutputStream*. El método *writeTo* permite indicar el fichero al que se transferirá el contenido del *array*. Una vez se ha terminado de escribir en el *array* los *bytes* a transferir, se ejecutará el método *flush()* que forzará a que sean transferidos.

```
try{
    FileOutputStream fos = new FileOutputStream("C:\\ficheros\\fich3.bin");
    ByteArrayOutputStream bos=new ByteArrayOutputStream();
    bos.write(74);
    bos.write((byte)'A');
    bos.write(86);
    bos.write(65);
    bos.writeTo(fos);
    bos.flush();
    bos.close();
}catch(IOException ioe){
    System.out.println(ioe);
}
```

5. *PrintStream:* permite escribir los datos de un *Stream* en otro automáticamente y sin necesidad de invocar el método *flush()*. Este método no lanzará excepciones *IOException*. Los métodos que se pueden emplear ya fueron descritos en el apartado 5.2.2. (Entrada y salida por consola).

```
try{
    PrintStream ps;
    FileOutputStream fos=new FileOutputStream("C:\\ficheros\\fich4.bin");
    // escribiendo en fichero
    ps=new PrintStream(fos);
    ps.println("Escribiendo en fichero ...");
    ps.printf("Escribiendo en fichero por %d ª vez ...\n",2);
    ps.close();
    fos.close();
    // escribiendo en consola
    ps=new PrintStream(System.out);
    ps.println("Escribiendo en consola ...");
    ps.printf("Escribiendo en consola por %d ª vez... \n",2);
}catch(IOException ioe){
    System.out.println(ioe);
}
```

```
run:
Escribiendo en consola ...
Escribiendo en consola por 2ª vez...
```

6. *FileInputStream:* posibilita leer *bytes* desde un fichero binario. Puesto que lo que se leen son *bytes*, se debe hacer un *cast* de los datos para poderlos manejar convenientemente.


```
try{
    FileInputStream fis=new FileInputStream("C:\\ficheros\\fich1.bin");
    int i;
    while((i=fis.read())!=-1){
        System.out.print((char)i);
    }
    fis.close();
}catch(IOException ioe){
    System.out.println(ioe);
}
```

```
run:
aeí
ou
```

7. *BufferedInputStream:* permite leer información de otro *InputStream* utilizando un *buffer* interno que mejora el rendimiento. El resultado final es el mismo que utilizando únicamente *FileInputStream*.

```
try{
    FileInputStream fis=new FileInputStream("C:\\ficheros\\fich1.bin");
    BufferedInputStream bis=new BufferedInputStream(fis);
    int i;
    while((i=bis.read())!=-1){
        System.out.print((char)i);
    }
    bis.close();
    fis.close();
}catch(IOException ioe){
    System.out.println(ioe);
}
```

```
run:
aeí
ou
```

8. *DataInputStream:* define los métodos *readBoolean*, *readByte*, *readUTF*..., que permiten leer datos de tipo primitivo desde un *InputStream*.

```
try{
    FileInputStream fis = new FileInputStream("C:\\ficheros\\fich2.bin");
    DataInputStream dis=new DataInputStream(fis);
    System.out.println(dis.readUTF());
    System.out.println(dis.readInt());
}catch(IOException ioe){
    System.out.println(ioe);
}
```

```
run:
cadena prueba
22
```

9. *ByteArrayInputStream:* permite usar un *array* de *bytes* como un *InputStream*.

```
byte[] buffer = {74, 65, 86, 65};
ByteArrayInputStream array = new ByteArrayInputStream(buffer);
int b = 0;
while ((b = array.read()) != -1) {
    System.out.println("El Carácter ASCII " + b + " es: " + (char) b);
}
```

```
run:
El Carácter ASCII 74 es: J
El Carácter ASCII 65 es: A
El Carácter ASCII 86 es: V
El Carácter ASCII 65 es: A
```

10. *SequenceInputStream:* flujo de entrada que puede combinar varios *InputStream* en uno.

```
try{
    byte[] buffer = (65, 80, 82, 69, 78, 68, 73, 69, 78, 68, 79, 32);
    ByteArrayInputStream io1 = new ByteArrayInputStream(buffer);
    FileInputStream io2 = new FileInputStream("C:\\ficheros\\fich3.bin");
    SequenceInputStream inst=new SequenceInputStream(io1, io2);
    int j;
    while((j=inst.read())!=-1){
        System.out.print((char)j);
    }
    System.out.println(" !!!");
    inst.close();
    io1.close();
    io2.close();
}catch(IOException ioe){
    System.out.println(ioe);
}
```

```
run:
APRENDIENDO JAVA !!!
```

11. *PipedOutputStream* y *PipedInputStream*: representan el concepto de *tubería* y son utilizados en programas multihilo. Permiten reproducir en un extremo (*PipedInputStream*) los datos que se están escribiendo en el extremo opuesto (*PipedOutputStream*).

Actividades propuestas

12.2. Existen más flujos de entrada-salida que los expuestos en el capítulo, pero una vez se comprende el funcionamiento de uno se puede entender fácilmente el del resto. Investiga sobre el flujo *AudioInputStream* y describe cuál es su uso. ¿Existe algún flujo cuya finalidad sea la opuesta?

12.3. Algunos de los flujos anteriormente descritos no heredan directamente de las clases *InputStream* y *OutputStream*. Identifícalos y construye un diagrama de clases que permita ver las relaciones jerárquicas entre las clases que representan los flujos descritos.

B) Clases para manejo de flujos de caracteres

Las clases que representen un flujo de entrada de caracteres heredarán de las clases abstractas *Reader* y *Writer*. Algunos de los métodos más importantes de estas clases son mostrados en los cuadros 12.3 y 12.4.

CUADRO 12.3
Métodos base de *Reader*

`int read()`	Lee el siguiente carácter desde el *InputStream*. Si retorna -1, indicará que no se pueden leer más *bytes*.
`long skip(n)`	Hace que se omitan los *n* primeros caracteres.
`void close()`	Cierra el *Reader*.

CUADRO 12.4
Métodos base de Writer

`void write(int c)`	Escribe un carácter.
`void write(String cadena)`	Escribe la cadena.
`void write(char [] array)`	Escribe el *array* de caracteres.
`void flush()`	Vacía el flujo de salida actual del *Writer*.
`void close()`	Vacía el flujo de salida actual del *Writer* y lo cierra.

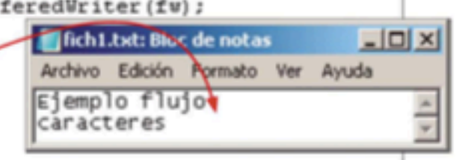
A continuación, se enumeran y describen las clases principales para el manejo de flujos. Se incluye también un ejemplo de uso para cada una de ellas:

1. *FileWriter:* permite escribir caracteres hacia un fichero de texto.

```
try{
    FileWriter fw = new FileWriter("C:\\ficheros\\fich1.txt");
    fw.write("Ejemplo flujo\r\n");
    fw.write("caracteres");
    fw.close();
}catch (IOException ioe){
    System.out.println(ioe);
}
```

2. *BufferedWriter:* posibilita escribir información de otro *Writer* utilizando un *buffer* interno que mejora el rendimiento. El método *newLine()* permite añadir un salto de línea y retorno de carro en la cadena.

```
try{
    FileWriter fw = new FileWriter("C:\\ficheros\\fich1.txt");
    BufferedWriter bw = new BufferedWriter(fw);
    bw.write("Ejemplo flujo");
    bw.newLine();
    bw.write("caracteres");
    bw.close();
    fw.close();
}catch (IOException ioe){
    System.out.println(ioe);
}
```



3. *CharArrayWriter:* permite usar un *array* de caracteres como *Writer*. El método *writeTo* permite indicar el *Writer* al que se transferirá el contenido del *array*.

```
try{
    CharArrayWriter caw=new CharArrayWriter();
    caw.write("JAVA");
    FileWriter fw=new FileWriter("C:\\ficheros\\fich2.txt");
    caw.writeTo(fw);
    fw.close();
}catch (IOException ioe){
    System.out.println(ioe);
}
```

fich2.txt: Bloc de notas
Archivo Edición Formato Ver Ayuda
JAVA

4. *PrintWriter:* es la implementación de la clase *Writer*. Permite escribir una cadena de caracteres en cualquier *OutputStream*. Añade métodos *print*, *printf* y *println* para imprimir contenidos de forma similar a como hace *System.out*, pero hasta que no se ejecuta el método *flush()* o *close()* los contenidos no son escritos.

```
try{
    PrintWriter pw;
    FileWriter fw = new FileWriter ("C:\\ficheros\\fich4.txt");
    // escribiendo en fichero
    pw = new PrintWriter(fw);
    pw.println("Escribiendo en fichero ...");
    pw.printf("Escribiendo en fichero por %d ª vez ...\n",2);
    pw.flush();
    pw.close();
    fw.close();
    // escribiendo en consola
    pw = new PrintWriter(System.out);
    pw.println("Escribiendo en consola ...");
    pw.printf("Escribiendo en consola por %d ª vez ...\n",2);
    pw.flush();
    pw.close();
}catch (IOException ioe){
    System.out.println(ioe);
}
```

fich4.txt: Bloc de notas
Archivo Edición Formato Ver Ayuda
Escribiendo en fichero ...
Escribiendo en fichero por 2ª vez ...

```
run:
Escribiendo en consola ...
Escribiendo en consola por 2ª vez ...
```

5. *StringWriter:* flujo de caracteres que recoge su salida en un *buffer* y que puede utilizarse para construir un *String*. El método *getBuffer()* devuelve el *buffer* contenedor de datos.

```
StringWriter sw = new StringWriter();
sw.write("Aprendiendo ");
sw.write("JAVA\n");
System.out.print(sw.toString());
System.out.print(sw.getBuffer());
```

```
run:
Aprendiendo JAVA
Aprendiendo JAVA
```


6. *FileReader:* flujo de caracteres destinado a la lectura de ficheros de texto.

```
try{
    FileReader fr = new FileReader("C:\\ficheros\\fich1.txt");
    int i;
    while((i=fr.read())!=-1)
        System.out.print((char)i);
    fr.close();
}catch (IOException ioe){
    System.out.println(ioe);
}
```

fich1.txt: Bloc de notas
Archivo Edición Formato Ver Ayuda
Ejemplo flujo caracteres

```
run:
Ejemplo flujo
caracteres
```

7. *BufferedReader:* permite leer información de otro *Reader* utilizando un *buffer* interno que mejora el rendimiento. Además del método *read()*, también dispone del método *readLine()*, que recupera una línea completa del *buffer*.

```
try{
    FileReader fr = new FileReader("C:\\ficheros\\fich1.txt");
    BufferedReader br = new BufferedReader (fr);
    System.out.println(br.readLine());
    int i;
    while((i=br.read())!=-1)
        System.out.print((char)i);
    br.close();
    fr.close();
}catch (IOException ioe){
    System.out.println(ioe);
}
```

fich1.txt: Bloc de notas
Archivo Edición Formato Ver Ayuda
Ejemplo flujo caracteres

```
run:
Ejemplo flujo
caracteres
```

8. *LineNumberReader:* se trata de un tipo especial de *BufferedReader* que lleva un conteo del número de línea que está leyendo en cada momento. El método *getLineNumber()* devuelve el número de línea inicial (por defecto comienza en 0), y el método *setLineNumber(n)* permite establecer a *n* el número de la línea actual.

```
try{
    FileReader fr = new FileReader("C:\\ficheros\\fich1.txt");
    LineNumberReader lnr = new LineNumberReader (fr);
    System.out.println(lnr.getLineNumber() + "-"+ lnr.readLine());
    lnr.setLineNumber(8);
    System.out.println(lnr.getLineNumber() + "-"+ lnr.readLine());
    lnr.close();
    fr.close();
}catch (IOException ioe){
    System.out.println(ioe);
}
```

fich1.txt: Bloc de notas
Archivo Edición Formato Ver
Ejemplo flujo caracteres

```
run:
0-Ejemplo flujo
8-caracteres
```

9. *CharArrayReader:* flujo de caracteres que recoge en un *buffer* los datos provenientes de un *array* de caracteres.

```
try{
    char []array = "123 JAVA".toCharArray();
    CharArrayReader car = new CharArrayReader(array);
    car.skip(4);
    int k = 0;
    while ((k = car.read()) != -1)
      System.out.print((char) k);
}catch (IOException ioe){
    System.out.println(ioe);
}
```

```
run:
JAVA
```

10. *StringReader:* flujo de caracteres que recoge en un *buffer* los datos provenientes de un *String*.

```
try{
    StringReader sr = new StringReader("Aprendiendo JAVA\n");
    int c=0;
    while((c=sr.read())!=-1)
        System.out.print((char)c);
}catch (IOException ioe){
    System.out.println(ioe);
}
```

```
run:
Aprendiendo JAVA
```

11. *PipedWriter* y *PipedReader:* similares a *PipedInputStream* y *PipedOutputStream* pero con flujos de caracteres.

Actividad propuesta 12.4

Busca información sobre los métodos *mark* y *reset* de las clases *Reader* y derivadas. Implementa un programa ejemplo en el que se vea su utilidad.

PARA SABER MÁS

En determinadas situaciones puede ser necesario transformar un flujo de *bytes* en un flujo de caracteres. Para ello Java ofrece las clases *InputStreamReader* y *OutputStreamWriter*, que convierten respectivamente un *InputStream* a *Reader* y un *OutputStream* a *Writer*.

12.3. Aplicaciones del almacenamiento de información en ficheros

Como se podrá imaginar, implementar aplicaciones o servicios que tengan que recibir todos los datos a través de las interfaces de entrada cada vez que estos se ejecutan tiene poca funcionalidad. En caso de manejarse grandes cantidades de información, esto implicaría un gran desperdicio de tiempo además de una potencial fuente de errores.

En su defecto los sistemas hacen uso de elementos de almacenamiento persistente desde los cuales se rescatan los datos para llevarlos hasta la memoria principal y, una vez ahí, ser manejados por los diferentes programas. Esta forma de trabajo tiene sus orígenes en la arquitectura de *Von Neumann*. Los elementos encargados de almacenar esa información son los *ficheros* y estos pueden almacenar contenidos de muy diverso tipo: texto, vídeo, imagen, ejecutables…

Asociado a esos contenidos está la *metainformación*, que es la información que permite que el contenido sea manejado correctamente por el sistema operativo y las aplicaciones. Dentro de esta *metainformación* se puede encontrar, entre otros, quién es el usuario propietario del fichero, el tamaño del fichero, los permisos de acceso, la fecha en la que fue creado…

En este capítulo el estudio se centrará en dos tipos bien diferenciados de ficheros: *ficheros de texto* (legibles para un humano) y *ficheros binarios* (legibles para sistemas o aplicaciones).

12.3.1. Tipos de ficheros. Registro

Para manejar correctamente un fichero desde un programa, es importante tener en cuenta el tipo de contenido que este almacenará:

- ✓ Si el contenido es *texto*, el acceso se realizará de forma *secuencial*. Esto implica que, si se necesita acceder al carácter contenido en la posición *i*, será necesario recorrer desde el primero hasta el *i-1*.
- ✓ Si el contenido es *binario*, el acceso podrá realizarse de forma secuencial o mediante acceso directo. El *acceso directo* permite llegar al *byte* ubicado en la posición *i* de forma directa y sin necesidad de recorrer todos los anteriores (de forma similar a como se accedería al elemento de un *array*).

En los ejemplos mostrados a lo largo del capítulo los accesos realizados a ficheros binarios han sido secuenciales y con datos simples (texto en la mayoría de los casos). Es posible, sin embargo, que los datos a leer o escribir se correspondan con los de objetos de una determinada clase y cuyo tamaño no sea precisamente el que ocupa un tipo de dato primitivo. Al conjunto de datos que permanecen juntos al ser leídos o escritos en un fichero se le conoce como *registro*. Por tanto, el conjunto de datos correspondiente a una instancia de clase insertada en un fichero binario será un registro.

12.3.2. Creación y eliminación de ficheros y directorios

Antes de proceder con la lectura y modificación de ficheros, es importante conocer los mecanismos de creación, eliminación y manejo de ficheros-directorios. La clase *File* proporciona la funcionalidad necesaria para ello. Sus constructores y métodos son descritos en los cuadros 12.5 y 12.6.

Cuadro 12.5
Constructores de la clase *File*

`File (String pathname)`	Constructor de objetos *File* a partir de un *path* completo.
`File (String padre, String hijo)`	Constructor de objetos *File* a partir de un *path* padre y el *path* hijo contenido en él.
`File (File padre, String hijo)`	Constructor de objetos *File* a partir de un objeto *File* existente y el *path* del hijo contenido.

Cuadro 12.6
Métodos de la clase *File*

`boolean createNewFile ()`	Crea un fichero vacío a partir de un objeto *File* existente, cuyo nombre es el *path* completo (únicamente si no existe previamente).
`boolean mkdir ()`	Construye un directorio a partir de un objeto *File* existente, cuyo nombre es el *path* completo si no existe previamente.
`boolean mkdirs ()`	Similar a *mkdir* pero creando todos los directorios intermedios del *path* que no existan.
`boolean delete ()`	Elimina el fichero o directorio identificado por el *path*.
`void deleteOnExit ()`	Fuerza a ejecutar un borrado del fichero o directorio identificado por el *path* cuando la máquina virtual finalice en condiciones normales.

A continuación, se muestra un ejemplo básico de uso de algunos de los métodos descritos:

```
try {
    File file1 = new File("C:\\ficheros\\Nuevo");
    file1.mkdir();
    File file2 = new File("C:\\ficheros","Nuevo1.bin");
    file2.createNewFile();
} catch (IOException ioe) {
    System.out.println(ioe);
}
```

A partir de la versión 1.7 Java incluyó la clase *Files* contenida dentro del paquete *java.nio*. Esta clase, cuyos métodos son todos estáticos, proporciona un repertorio más completo que el ofrecido por la clase *File* para crear, eliminar y manejar ficheros. Dentro del mismo paquete se encuentran las clases *Path* que permite localizar un fichero en un sistema de ficheros y la clase *Paths*, que permite obtener objetos de tipo *Path* y cuyos métodos son también estáticos.

Actividad propuesta 12.5

Elabora una tabla comparativa entre las clases *Files* y *File* en la que se muestre la equivalencia entre los métodos de una clase y otra para el borrado y creación de ficheros-directorios.

Sabías que...

Además de los métodos de las clases *File* y *Files*, también es posible crear los ficheros a la vez que se accede a ellos para escribir información.

No todos los sistemas operativos utilizan los mismos caracteres para separar los directorios que forman un *path* y puede que tampoco lo hagan para separar los diferentes *Path* representados en una misma cadena. La clase *File* permite conocer los elementos empleados para ello a partir de la información de los campos estáticos *pathSeparator* y *separator*.

Investiga

Busca información sobre la clase *Files* y los métodos que ofrece para la creación y borrado de ficheros y directorios. Propón un ejemplo de uso en el que se utilicen algunos de ellos.

12.3.3. Apertura y cierre de ficheros. Modos de acceso

Independientemente de cómo se lleve a cabo la creación del fichero, es importante saber cómo abrirlo para acceder al contenido y cómo cerrarlo una vez se haya terminado de utilizar. Dependiendo del tipo de fichero y del tipo de acceso se emplearán unas clases u otras.

A) Acceso secuencial

En caso de realizar accesos secuenciales se pueden utilizar las clases *FileReader*, *FileWriter*, *FileInputStream* y *FileOutputStream*, así como las clases *Buffered* relacionadas con estas (descritas en el apartado 12.2.4). Para indicar cómo acceder al fichero, será suficiente con invocar al constructor correspondiente.

Cuadro 12.7
Constructores de *FileReader*, *FileWriter*, *FileInputStream* y *FileOutputStream*

`FileReader (String path)`
`FileReader (File file)`
`FileInputStream (String path)`
`FileInputStream (File file)`
`FileWriter (String path [, boolean append])`
`FileWriter (File file [, boolean append])`
`FileOutputStream (String path [, boolean append])`
`FileOutputStream (File file [, boolean append])`

En caso de que el fichero sea abierto para escritura con *FileWriter* o *FileOutputStream*:

- Si el fichero no existe, será creado y posteriormente se abrirá para realizar la escritura.
- Si el fichero existe, se abrirá para escritura y:
  - si el argumento *append* se establece a *true*, el contenido se añadirá al final del fichero.
  - si el argumento *append* se establece a *false* o es omitido, el fichero será sobrescrito.

Para realizar el cierre del fichero será suficiente con invocar al método *close()*.

- **Files**

La clase *Files* ofrece métodos que permiten obtener objetos *BufferedReader* y *BufferedWriter* con los que realizar el acceso secuencial a ficheros de texto plano, y objetos *InputStream* y *OutputStream* con los que realizar el acceso secuencial a ficheros binarios. Permite, además, especificar el modo de acceso al fichero mediante *opciones*. Los métodos utilizados para obtener estos flujos de acceso a ficheros son descritos en el cuadro 12.8.

CUADRO 12.8
Métodos de la clase *Files*

`BufferedReader newBufferedReader (Path path [, Charset cs])`
`BufferedWriter newBufferedWriter (Path path [, Charset cs][, OpenOption opcion1 …, OpenOption opcionN])`
`InputStream newInputStream (Path path [, OpenOption opcion1 …, OpenOption opcionN])`
`OutputStream newOutputStream (Path path [, OpenOption opcion1 …, OpenOption opcionN])`

en donde:

- ✓ *path* es un objeto de la clase *Path* que permite ubicar los ficheros en un sistema de archivos.
- ✓ *cs* hace referencia al conjunto de caracteres a emplear. En caso de omitirse, por defecto se establece a *StandardCharsets.UTF_8*.
- ✓ cada opción *OpenOption* puede ser cualquier constante del enumerado *StandardOpenOption*.

SABÍAS QUE...

Si no se establecen opciones cuando se invoca a *Files.newBufferedWriter*, por defecto se establecen las opciones CREATE, TRUNCATE_EXISTING y WRITE.

CUADRO 12.9
Valores del enumerado *StandardOpenOption*

`APPEND`	Si el fichero es abierto para escritura, los *bytes* son añadidos al final del fichero.
`CREATE`	Crea un fichero nuevo en caso de que no exista.
`CREATE_NEW`	Crea un fichero nuevo en caso de que no exista y lanza una excepción en caso de que ya existiera.
`DELETE_ON_CLOSE`	Elimina el fichero al cerrarlo.
`DSYNC`	Fuerza que cada actualización del contenido del archivo se escriba de forma síncrona en el dispositivo de almacenamiento subyacente.
`READ`	Abre el fichero para lectura.
`SPARSE`	Cuando se usa con la opción CREATE_NEW, proporciona una sugerencia de que el nuevo archivo será un archivo *disperso*. La opción se ignora cuando el sistema de archivos no admite la creación de archivos dispersos.
`SYNC`	Fuerza que cada actualización del contenido del archivo o de sus metadatos se escriba de forma síncrona en el dispositivo de almacenamiento subyacente.
`TRUNCATE_EXISTING`	Si el fichero ya existe y es abierto para escritura, lo sobrescribe.
`WRITE`	Abre el fichero para escritura.

INVESTIGA

Indaga sobre los diferentes conjuntos de caracteres disponibles y propón un escenario en el que sea lógico el uso de uno de ellos.

B) Acceso directo

Si el acceso a realizar es un acceso directo a un fichero binario, se deberá hacer uso de la clase *RandomAccessFile*. Se puede ver la sintaxis de los constructores en el cuadro 12.10.

CUADRO 12.10
Constructores de la clase *RandomAccessFile*

`new RandomAccessFile(String path, String modo);`
`new RandomAccessFile(File file, String modo);`

Para construir un objeto *RandomAccessFile* se podrá emplear directamente una cadena que represente el *path* del fichero o bien un objeto *File*. Además, será imprescindible indicar el modo de acceso.

CUADRO 12.11
Modos de acceso para *RandomAccessFile*

`r`	Abrir para lectura. Al invocar cualquier método de escritura del objeto resultante se generará una excepción *IOException*.
`rw`	Abrir para leer y escribir. Si el archivo no existe, intentará crearlo.
`rwd`	Abrir para leer y escribir, como con "rw", pero requiere que cada actualización del contenido del archivo se escriba de forma síncrona en el dispositivo de almacenamiento subyacente.
`rws`	Similar a "rwd", pero también forzará a que se escriba de forma síncrona en el dispositivo de almacenamiento subyacente cualquier cambio producido en los metadatos del archivo.

PARA SABER MÁS

El modo de acceso a un fichero estará supeditado a los permisos establecidos en el sistema operativo anfitrión sobre el cual se ejecuta el programa.

12.3.4. Escritura y lectura de información en ficheros

La forma de llevar a cabo la lectura o escritura de ficheros dependerá tanto del tipo de información a almacenar como del tipo de acceso a llevar a cabo.

A) Acceso secuencial

Para realizar el acceso secuencial a ficheros de texto o binarios, se puede hacer uso de los métodos *read* y *write* de los diferentes flujos de entrada y salida. Para visualizar ejemplos de lectura y escritura secuenciales puede consultarse el apartado 12.2.4.

- **Files**

Si se desea utilizar la clase *Files* para la lectura o escritura secuencial de ficheros, se podrá disponer de dos alternativas:

1. Utilizar los métodos de *Files* para realizar la escritura y lectura de información (cuadro 12.12).

CUADRO 12.12
Métodos de lectura y escritura de la clase *Files*

`Path write (Path path, byte[] bytes [,OpenOption op1 … [, OpenOption opN])`	Escribe los *bytes* en el fichero representado por *path*.
`byte[] readAllBytes (Path path)`	Lee todos los *bytes* del fichero representado por *path* y los devuelve en un *array* de *bytes*.
`List<String> readAllLines (Path path [, Charset cs])`	Devuelve una colección de *String* correspondiente a las líneas del fichero representado por *path*.

```
try{
    // Escritura
    Path fichero = Paths.get("C:\\ficheros\\Nuevo2.bin");
    byte [] texto1 = (65, 80, 82, 69, 78, 68, 73, 69, 78, 68, 79, 13, 10);
    byte [] texto2 = (74, 65, 86, 65, 13, 10);
    Files.write(fichero, texto1);
    Files.write(fichero, texto2, StandardOpenOption.APPEND);
    // Lectura en array de bytes
    byte [] contenido = Files.readAllBytes(fichero);
    for (int i=0; i<contenido.length; i++)
        System.out.print((char) contenido[i]);
    // Lectura en colección
    List<String> lineas = Files.readAllLines (fichero);
    System.out.print(lineas.toString());
}catch (IOException ioe){
    System.out.println(ioe);
}
```

```
run:
APRENDIENDO
JAVA
[APRENDIENDO, JAVA]
```

2. Recuperar el objeto *BufferedReader*, *BufferedWriter*, *InputStream* u *OutputStream* correctamente configurado con el modo de acceso y utilizarlo conforme a lo descrito en el apartado 12.2.4. En el siguiente ejemplo se muestra un programa que lee un fichero con contenido *Lorem ipsum* y lo escribe en otro fichero reemplazando la secuencia de caracteres *dui* por ***:

```
try {
    Path input = Paths.get("C:\\ficheros\\FichOrigen.txt");
    Path output = Paths.get("C:\\ficheros\\FichDestino.txt");
    BufferedReader br = Files.newBufferedReader(input, Charset.defaultCharset());
    BufferedWriter bw = Files.newBufferedWriter(output, Charset.defaultCharset(),
       StandardOpenOption.WRITE , StandardOpenOption.CREATE ,
       StandardOpenOption. TRUNCATE_EXISTING);
    String linea;
    while ((linea = br.readLine ()) != null ) {
        if (linea.contains("dui"))
            linea = linea.replaceAll("dui", "****");
        bw.write(linea , 0, linea.length ());
        bw.newLine ();
    }
    br.close ();
    bw.close ();
}catch (IOException ioe){
    System.out.println(ioe);
}
```

FichDestino.txt: Bloc de notas

Archivo Edición Formato Ver Ayuda

```
Lorem ipsum dolor sit amet consectetur
adipiscing elit nascetur vestibulum urna
vehicula, quisque nibh *** lectus aptent ***
pulvinar eu taciti quis condimentum, parturient
suscipit mattis fringilla fermentum dignissim
class malesuada turpis tempus. cras scelerisque
eros luctus ante blandit habitasse integer
etiam suspendisse pretium, *** id pulvinar
```

B) Acceso directo

Para realizar el acceso directo a un fichero binario se empleará la clase *RandomAccessFile*. En el cuadro 12.13 se describen algunos de sus métodos.

CUADRO 12.13
Métodos de acceso de la clase *RandomAccessFile*

`void close ()`	Cierra el fichero y libera los recursos asociados.
`int readInt ()`	Lee un entero con signo de 32 bits.
`String readUTF ()`	Lee una cadena de este archivo.
`void seek (long pos)`	Desplaza el puntero del archivo a la posición *pos* (desde el principio del archivo).
`int skipBytes(int n)`	Hace que se omitan los *n* primeros *bytes*.
`void writeDouble (double v)`	Convierte el argumento *double* a un *long* y lo escribe con ocho *bytes*.
`void writeFloat (float v)`	Convierte el argumento *float* a un *int* y lo escribe con cuatro *bytes*.
`void write (int b)`	Escribe el *byte* especificado en este archivo.
`int read ()`	Lee un *byte* del archivo.
`long length ()`	Devuelve la longitud del archivo.

Es importante tener en cuenta que Java utiliza caracteres Unicode y, por este motivo, cada caracter ocupará realmente 2 *bytes*. Para conocer el espacio que ocupa cada tipo de dato primitivo se puede consultar el cuadro 4.2 del capítulo 4, que recoge los tipos de datos primitivos. A continuación, se muestra un ejemplo básico de uso de la clase *RandomAccessFile* en el que se escriben contenidos en un fichero binario para posteriormente sobrescribir algunos accediendo directamente a su posición.

```
File f = new File("C:\\ficheros\\Prueba.bin");
f.delete();
RandomAccessFile raf = new RandomAccessFile("C:\\ficheros\\Prueba.bin", "rw");
String s = "0123456789";
raf.writeChars(s);
raf.seek(0);
for (int i = 0; i < raf.length()/2; i++) {
    System.out.print("" + raf.readChar());
}
System.out.println();
raf.seek(8);
raf.writeChars("**");
raf.writeChars("**");
raf.seek(0);
for (int i = 0; i < raf.length()/2; i++) {
    System.out.print("" + raf.readChar());
}
System.out.println();
```

```
run:
0123456789
0123**6789
```


Aunque en el ejemplo anterior los datos almacenados han sido simples caracteres, también es posible almacenar datos de elementos más complejos como puede ser un objeto. Para poder realizar una lectura posterior de estos, será necesario que todas las instancias cuyos datos se vayan a escribir tengan longitud fija.

Actividad propuesta 12.6

Busca información sobre cómo establecer la longitud fija de un campo cadena mediante el empleo de la clase *StringBuffer*. Propón un ejemplo de uso en el que se almacenen datos de objetos de una clase en un fichero binario y posteriormente puedan ser recuperados.

12.3.5. Almacenamiento de objetos en ficheros. Serialización

Hasta ahora se han mostrado varios mecanismos para hacer persistentes los datos manejados en un programa. Sin embargo, todos ellos implican que, para recuperar la información una vez ha sido volcada al fichero, se necesita conocer el orden de lectura y la estructura con la que fue almacenada.

Java proporciona una forma mucho más cómoda de almacenar la información referente a objetos de una clase en ficheros mediante la transformación de estos a un conjunto de *bytes*. Este proceso de transformación se conoce como *serialización*.

Para que el objeto de una clase sea serializable al completo se debe cumplir:

1. Que la clase a la cual pertenece implemente la interfaz *java.io.Serializable*. Si se intenta serializar un objeto cuya clase no implementa *java.io.Serializable*, se provocará la excepción *NotSerializableException*.
2. Que todos los campos de la clase sean de tipo primitivo o, en caso de tratarse de una clase compuesta por otras, clases *serializables*.
3. Que la clase declare un campo estático y privado *serialVersionUID*. Se trata de un número de versión que posee cada clase serializable y que es usado en el proceso de *deserialización* para verificar que tanto emisor como receptor del objeto serializado mantienen una compatibilidad en cuanto a serialización se refiere con respecto a las clases que tienen cargadas. Si no se declara explícitamente, se hará de forma automática en tiempo de ejecución, pero dado que se calcula teniendo en cuenta varios aspectos de la clase que pueden variar dependiendo de la implementación del compilador, es posible que se produzca una *InvalidClassException* durante el proceso de *deserialización*.

En el supuesto de que haya datos que no se desee serializar, estos deberán ser declarados con el modificador *transient*. Asimismo, los campos *static* tampoco serán serializados.

Para manejar los objetos serializables se hará uso de las clases:

- *ObjectOutputStream* y su método *writeObject* para la escritura (*serialización*).

- *ObjectInputStream* y su método *readObject* para la lectura (*deserialización*). El resultado de ejecutar el método *readObject* es un objeto de la clase base *Object*, por lo que será necesario realizar el *casting* adecuado.

A continuación, puede verse un ejemplo de uso:

```
package lecturaescritura;
import java.io.Serializable;

public class ObjetoSerializable implements Serializable {
    private static final long serialVersionUID = 1L;
    static int campo1;
    int campo2;
    transient int campo3;

    ObjetoSerializable(int c1, int c2, int c3){
        campo1 = c1;
        campo2 = c2;
        campo3 = c3;
    }
    ObjetoSerializable(){
        campo1 = 7;
        campo2 = 7;
        campo3 = 7;
    }
}
```

```
try {
    // Serializar un objeto a un archivo
    FileOutputStream fos=new FileOutputStream("C:\\ficheros\\Serializado.bin");
    ObjectOutputStream oos = new ObjectOutputStream(fos);
    ObjetoSerializable objeto1 = new ObjetoSerializable(1, 2, 3);
    oos.writeObject(objeto1);
    oos.close();
    // Leer un objeto serializado desde un archivo
    FileInputStream fis = new FileInputStream("C:\\ficheros\\Serializado.bin");
    ObjectInputStream ois = new ObjectInputStream(fis);
    ObjetoSerializable objeto2 = new ObjetoSerializable();
    System.out.println(objeto2.campo1 +" "+objeto2.campo2+" "+objeto2.campo3);
    objeto2 = (ObjetoSerializable)ois.readObject();
    System.out.println(objeto2.campo1 +" "+objeto2.campo2+" "+objeto2.campo3);
    ois.close();
}catch (IOException | ClassNotFoundException ioe) {
    System.out.println(ioe);
}
```

```
run:
7 7 7
7 2 0
```

Sabías que...

Dado que un objeto serializable es un conjunto de bytes, este puede ser transmitido de un equipo a otro a través de la red.

Las colecciones Java estudiadas en el capítulo 9 implementan la interfaz *java.io.Serializable* y, por tanto, es muy sencillo almacenar o transmitir toda la información contenida en ellas.

Investiga

Es posible controlar la serialización y mejorar el rendimiento de este proceso mediante el empleo de la interfaz *Externalizable*. Investiga sobre su uso y propón un programa ejemplo en el que se utilice.

12.3.6. Utilización de los sistemas de ficheros

Además de crear, eliminar, leer y escribir ficheros o directorios, existen otras tareas que se realizan habitualmente sobre un sistema de ficheros y para las que *Java* ofrece soporte. En el cuadro 12.14 puede ver un listado de los métodos ofrecidos por la clase *File*.

Cuadro 12.14
Métodos de la clase *File* para el manejo de sistemas de ficheros

`boolean exists()`	Devuelve *true* si el *path* existe, y *false* en caso contrario.
`long length()`	Devuelve la longitud del fichero.
`boolean renameTo (File dest)`	Renombra o mueve el fichero o directorio representado a *dest*.
`boolean isFile()`	Devuelve *true* si el objeto *File* desde el que se invoca representa a un fichero.
`boolean isDirectory()`	Devuelve *true* si el objeto *File* desde el que se invoca representa a un directorio.
`String[] list()`	Devuelve un listado de los nombres de ficheros contenidos en un directorio en un *array* de *Strings*.
`File[] listFiles()`	Obtiene un *array* de objetos *File* que representan a cada uno de los ficheros contenidos en un directorio.
`String getName()`	Devuelve el nombre del fichero o directorio.
`String getParent()`	Devuelve el nombre del directorio padre del fichero o directorio.
`File getParentFile()`	Devuelve el objeto *File* correspondiente al directorio padre del fichero o directorio.
`String getPath()`	Devuelve el *path*.

`boolean canRead()`
`boolean canWrite()`
`boolean canExecute()`
	Devuelven *true* en caso de que el fichero pueda ser abierto para lectura, escritura o ejecución (respectivamente), y *false* en caso contrario.

`boolean setReadable (boolean ejecutable [,boolean propietario])`
`boolean setWritable (boolean ejecutable [,boolean propietario])`
`boolean setExecutable (boolean ejecutable [,boolean propietario])`
	Establecen los permisos de lectura, escritura o ejecución (respectivamente) para todos los usuarios del sistema o, únicamente, para el usuario propietario.

Investiga

a) La clase *Files* también ofrece métodos para establecer la visibilidad del fichero, así como los permisos de lectura, escritura y ejecución. Investiga sobre estos métodos y propón un programa ejemplo en el cual se establezcan distintos permisos de acceso y niveles de visibilidad.

b) Investiga sobre los métodos ofrecidos por la clase *Files* para llevar a cabo el manejo de la metainformación de un fichero. Propón un ejemplo de uso.

Resumen

- Todo proceso de comunicación usuario-programa se realiza mediante flujos de información.
- En Java existen muchos tipos de flujos de información y cada uno está destinado a un tipo de datos.
- Los datos manejados por un programa son volátiles, a menos que sean almacenados en ficheros sobre un sistema de archivos.
- Para almacenar los datos del programa en ficheros también se emplean flujos de datos.
- Existe un amplio repertorio de clases y métodos para el manejo de ficheros.
- Almacenar datos primitivos en un fichero es una tarea sencilla. Sin embargo, almacenar datos complejos correspondientes a objetos dificulta el proceso de recuperación de su información.
- Se denomina registro a cada conjunto de datos del fichero que se corresponda con la información de un objeto u otra estructura.
- El concepto *serialización* facilita enormemente la tarea de almacenaje en ficheros del contenido de objetos.

Ejercicios propuestos

1. ¿Es posible utilizar un fichero binario para almacenar texto? Justifica tu respuesta.
2. ¿Qué consecuencias puede tener el no cerrar un *Stream* cuando se ha terminado de utilizar?
3. Las opciones *rwd* y *rws* empleadas con la clase *RandomAccessFile*, ¿ralentizan o aceleran la escritura de información en un fichero? Justifica tu respuesta.
4. ¿Qué es la metainformación de un fichero? ¿Puede manejarse desde un programa Java?
5. Si tuvieras que leer un fichero de texto cuyo contenido está escrito en chino, ¿utilizarías el conjunto de caracteres *StandardCharsets.ISO_8859_1*? Justifica tu respuesta.
6. ¿Es posible sustituir el método *seek* por el método *skipBytes*, ambos de la clase *RandomAccessFile*, en algún caso? Justifica tu respuesta.

Supuestos prácticos

1. Implementa un programa que lea los datos de un fichero de texto y que vuelque en otro fichero todas las líneas del primero cuyo contenido esté en mayúsculas.
2. Crea un programa que vuelque en un fichero de texto la serie de números del 0 al 100 (uno por línea) y que, a continuación, lea cada número desde el fichero para calcular la suma de todos los números y los muestre por pantalla.
3. Se desea implementar un programa para gestionar la información de los alumnos de un colegio. Entre los datos a almacenar de cada alumno está el nombre (*String* de un tamaño no superior a 30 caracteres), los apellidos (*String* de un tamaño no superior a 30 caracteres para cada apellido) y la edad (número entero representado mediante un tipo *byte*). Ten en cuenta que:
   - El programa permitirá añadir nuevos alumnos, así como consultar, modificar y eliminar los datos del alumno *i-ésimo*.
   - La información de cada alta, baja o modificación debe ser persistente, lo que implica que debe realizarse sobre el fichero.
   - Además de las funciones anteriormente citadas, también estará la de obtener un listado de los alumnos ordenado según fueron dados de alta. Las únicas operaciones que pueden alterar ese orden son las altas o bajas.

   Para garantizar la persistencia de los datos utiliza serialización.
4. A partir del supuesto práctico anterior, realiza las modificaciones necesarias para mantener la funcionalidad del programa sin utilizar serialización. En este caso el programa realizará accesos directos a los registros de cada alumno que sean necesarios.
5. Implementa un programa que reciba en sus parámetros de entrada el nombre de un directorio y la extensión de un tipo de fichero. A continuación, mostrará los nombres de los archivos contenidos en dicho directorio cuya extensión coincida con la indicada.
6. Implementa un programa que reciba las rutas de dos directorios y realiza la copia de todo el contenido del primero (subdirectorios incluidos) en el segundo.

Actividades de autoevaluación

1. ¿Qué resultado producirá el siguiente fragmento de código?

```
package lecturaescritura;
import java.io.*;
public class LecturaEscritura {
    public static void main(String[] args){
        PrintWriter pw;
        pw = new PrintWriter(System.out);
        pw.printf("Escribiendo en consola por %d ª vez ...\n",1,3,4,5);
    }
}
```

- ☐ *a)* Error de compilación, pues el número de argumentos pasados a *printf* es superior al indicado en la cadena de formato.
- ☐ *b)* El código compilará y la ejecución imprimirá "Escribiendo en consola por 1ª vez ...", terminando con un salto de línea.
- ☐ *c)* El código compilará y la ejecución imprimirá "Escribiendo en consola por 1ª vez ...\n".
- ☐ *d)* El código compilará y la ejecución no imprimirá nada por consola.

2. Las clases *BufferedXXXX*:
   - ☐ *a)* Hacen uso de un *buffer* intermedio.
   - ☐ *b)* Controlan los accesos a los ficheros para mejorar el rendimiento de la operación.
   - ☐ *c)* Las opciones *a)* y *b)* son correctas.
   - ☐ *d)* Todas las opciones anteriores son incorrectas.

3. ¿En qué paquete se encuentra la clase *Files*?
   - ☐ *a)* *java.lang.*
   - ☐ *b)* *java.io.*
   - ☐ *c)* *java.nio.*
   - ☐ *d)* *java.util.*

4. ¿Qué método emplearía para crear un fichero vacío a partir de un objeto *File* existente sobre el cual se ha establecido el *path* completo?
   - ☐ *a)* *newFiles()*
   - ☐ *b)* *newFile()*
   - ☐ *c)* *createFile()*
   - ☐ *d)* *createNewFile()*

5. ¿Cuál de las siguientes clases no hereda directamente de *Reader*?
   - ☐ *a)* *CharArrayReader.*
   - ☐ *b)* *LineNumberReader.*
   - ☐ *c)* *BufferedReader.*
   - ☐ *d)* *StringReader.*

6. ¿Cuál de las siguientes opciones de los flujos creados con *Files* fuerza que cada actualización del contenido del archivo se escriba de forma síncrona en el dispositivo de almacenamiento subyacente?
   - ☐ *a)* DSYNC.
   - ☐ *b)* SYNC.
   - ☐ *c)* Las opciones *a)* y *b)* son correctas.
   - ☐ *d)* Todas las opciones anteriores son incorrectas.

7. ¿Cuál de los siguientes modos de *RandomAccessFile* fuerza que cada actualización del contenido del archivo se escriba de forma síncrona en el dispositivo de almacenamiento subyacente?
   - ☐ *a)* r y rw.
   - ☐ *b)* rw y rwd.
   - ☐ *c)* rwd y rws.
   - ☐ *d)* Todas las opciones son incorrectas.

8. El método de *RandomAccessFile*, que permite desplazar el puntero del archivo a una posición determinada para proceder con la lectura o escritura desde dicha posición, es:
   - ☐ *a)* *seek.*
   - ☐ *b)* *skipBytes.*
   - ☐ *c)* *flush.*
   - ☐ *d)* *desp.*

9. La clase empleada para manejar flujos de datos de entrada binarios es:
   - ☐ *a)* *InputStream.*
   - ☐ *b)* *OutputStream.*
   - ☐ *c)* *StreamInput.*
   - ☐ *d)* *InputReader.*

10. ¿Cuál de las siguientes no es una clasificación válida de los flujos de datos?
    - ☐ *a)* Según la dirección del flujo de datos.
    - ☐ *b)* Según tipo de acceso a la información.
    - ☐ *c)* Según el tipo de información manejada.
    - ☐ *d)* Todas las opciones anteriores son correctas.

SOLUCIONES:

1.	a	b	c	**d**	5.	a	**b**	c	d						
2.	a	b	**c**	d	6.	a	b	**c**	d	9.	**a**	b	c	d	
3.	a	b	**c**	d	7.	a	b	**c**	d	10.	a	b	c	**d**	
4.	a	b	c	**d**	8.	**a**	b	c	d						

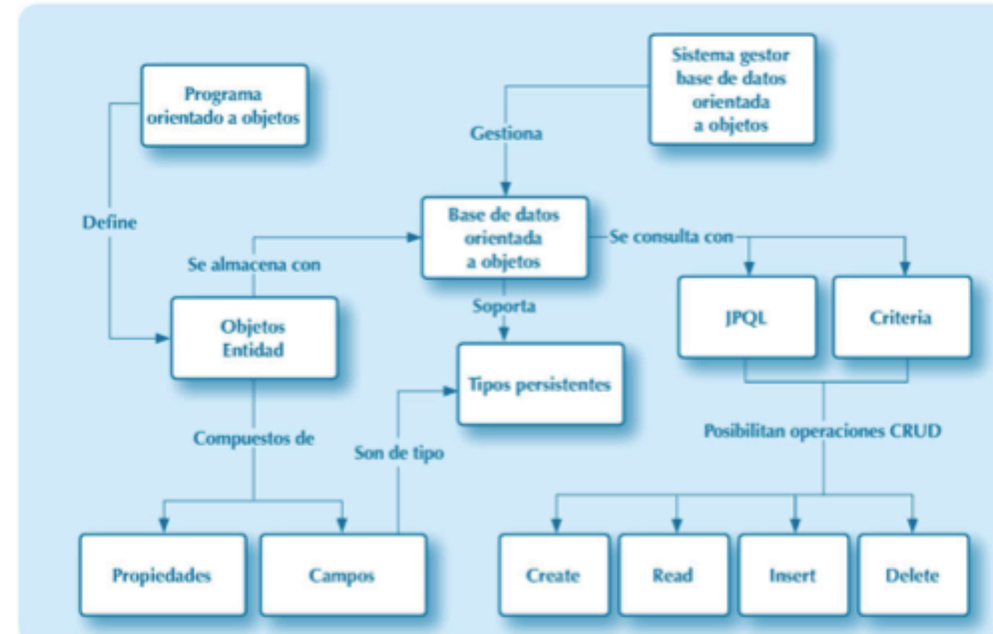
13

Mantenimiento de la persistencia de los objetos

Objetivos

- ✓ Conocer las características de las bases de datos orientadas a objetos.
- ✓ Instalar un sistema gestor de bases de datos orientadas a objetos.
- ✓ Crear bases de datos orientadas a objetos.
- ✓ Aprender a conectar con las bases de datos orientadas a objetos.
- ✓ Dominar los diferentes tipos de datos soportados en una base de datos orientada a objetos.
- ✓ Manejar el lenguaje de definición de objetos empleado en bases de datos orientadas a objetos (JPQL).
- ✓ Saber cómo realizar consultas contra una base de datos orientada a objetos.

Mapa conceptual



Glosario

API. Es la abreviatura de *Application Programming Interface* y representa al conjunto de funciones, procedimientos y especificaciones que una aplicación ofrece para que otras puedan comunicarse con ella.

Base de datos. Conjunto de datos almacenados de forma organizada para su posterior uso.

BDOO. También conocida como OODB, son las siglas de "base de datos orientada a objetos".

Clave primaria. También conocida como *Primary Key o PK*. En bases de datos orientadas a objetos se le denomina así al campo, o grupo de campos, que identifica de forma unívoca a cada objeto perteneciente a una clase.

Commit. En un sistema gestor de base de datos se refiere a la acción de confirmar una operación haciendo los cambios persistentes.

CRUD. Es el conjunto típico de operaciones a realizar sobre una base de datos y que consisten en inserción (*Create*), recuperación (*Read*), modificación (*Update*) y borrado (*Delete*).

GPL. GNU General Public License es una licencia muy utilizada en software libre de código abierto que garantiza la libertad de uso, estudio, copia y modificación del software.

JavaBeans. Se trata de un modelo de componentes que permite encapsular objetos en otros para ser manejados de forma conjunta.

Join. Sentencia empleada en bases de datos que permite combinar registros varias tablas o entidades.

JPA. También conocida como *Java Persistence API*, es la API desarrollada para la plataforma Java EE que proporciona los mecanismos para gestionar la persistencia de objetos en bases de datos orientadas a objetos.

JPQL. Sigla de *Java Persistence Query Language*, es un tipo de lenguaje de consultas para bases de datos orientadas a objetos independiente de la plataforma.

ODBMS. Es la sigla de *Object Database Management System* (gestor de base de datos orientada a objetos). Integra bases de datos con un lenguaje de programación orientado a objetos.

POJO. Es la sigla de *Plain Old Java Object* y hace referencia a los objetos de aquellas clases cuyos atributos son privados y accedidos mediante los métodos *get* y *set* públicos correspondientes.

Query. Se trata de una cadena de texto cuyo contenido se corresponde con el de una consulta, inserción, modificación o borrado para realizar en una base de datos.

SQL. Sigla de *Structured Query Language*. Es un lenguaje estándar de acceso a bases de datos relacionales que permite la selección, inserción, modificación y eliminación de datos en sistemas gestores de bases de datos.

TB. Sigla de *terabyte* (10^{12} *bytes*).

Transacción. Conjunto de instrucciones que deben llevarse a cabo de forma atómica.

13.1. Introducción

Una vez se ha tomado conciencia de la importancia de garantizar la persistencia de la información manejada por un programa y se ha tenido una primera experiencia con el manejo de ficheros, ha llegado el momento de dar un paso más y aprender a utilizar un sistema más robusto y eficiente.

Cuando la cantidad de datos a gestionar crece y la estructura de estos se complica, el uso de sistemas de ficheros se convierte en una tarea compleja y susceptible de provocar errores. Para este tipo de escenarios se dispone de las bases de datos que ofrecen mejores resultados y son más fáciles de manejar. En este capítulo se someterán a estudio las bases de datos orientadas a objetos.

13.2. Base de datos orientada a objetos

Una *base de datos orientada a objetos* (*BDOO* u *OODB*) es un sistema organizado de datos que representa la información en forma de objetos, entendiéndose el término objeto de forma similar a como se hace en el paradigma de programación orientada a objetos.

Este tipo de bases de datos se hicieron muy conocidas en la década de los noventa, aunque finalmente no tuvieron el impacto esperado y terminaron siendo superadas por las bases de datos relacionales. Recientemente han vuelto a suscitar un gran interés entre la comunidad de desarrolladores y administradores de bases de datos.

13.3. Características de las bases de datos orientadas a objetos

Una base de datos orientada a objetos difiere en muchos aspectos de las tradicionales bases de datos relacionales, aunque también tienen bastantes similitudes. A continuación, se describen algunas de las características de estos sistemas:

- La mayoría incluye algún tipo de lenguaje para realizar consultas.
- El acceso a los objetos es más rápido, ya que pueden ser recuperados directamente sin una búsqueda, simplemente siguiendo los punteros y referencias.
- Utilizan la misma definición de tipos que el lenguaje de programación.
- Son diseñadas para trabajar junto con lenguajes de programación orientados a objetos (Java, C#,Visual Basic.NET y C++).
- A diferencia de los sistemas de bases de datos relacionales, no es necesario definir tablas o convertir objetos en registros y viceversa usando SQL, sino que pueden ser insertados o recuperados de la base de datos directamente.

13.4. Instalación del gestor de bases de datos

Aunque existen varias alternativas, *ObjectDB* es un candidato idóneo por ser un sistema de gestión de base de datos orientado a objetos (ODBMS) especialmente diseñado para Java. Proporciona todos los servicios estándar de administración de bases de datos (almacenamiento, recuperación, transacciones, administración de bloqueos, procesamiento de consultas…), pero utilizando un modelo orientado a objetos.

Aunque es un software comercial, se puede descargar y utilizar sin coste con ciertas restricciones que no dificultan su uso para pequeños proyectos, tareas académicas y entornos de evaluación o aprendizaje (un máximo de 10 clases de entidad y un millón de objetos de entidad por cada archivo de base de datos). Soporta bases de datos de hasta 128 TB.

Recurso web

A través de la URL https://www.objectdb.com/download podrás acceder a la página oficial de descargas de *ObjectDB* y obtener la última versión estable del producto.

Para realizar la instalación, una vez descargado, se deberá descomprimir en una carpeta que sea accesible a los proyectos en los que será utilizado.

Una vez descomprimido, se encontrarán los directorios contenedores de las bases de datos, el explorador de entidades desde el que se podrá realizar la gestión de las diferentes entidades, el programa servidor para permitir aceptar conexiones desde otros equipos cliente, el fichero de configuración de *ObjectDB*…

Figura 13.1
Contenido del directorio de instalación de *ObjectDB*.

Actividades propuestas

13.1. Descarga e instala el sistema gestor de base de datos *ObjectDB*.
13.2. ¿Para qué se utiliza el paquete *converter.jar*?

Para saber más

Otra alternativa es *DB4O (DB for Objects)*, que fue distribuido bajo licencia *GPL* y también comercial. Fue absorbida por *Versant (Actian)* y la versión libre a día de hoy ya no está siendo actualizada, aunque aún es posible descargar y utilizar libremente la versión *db4o-8.0* desde la URL https://www.dropbox.com/s/s6farl3dqez7e1r/db4o-8.0.276.16149-java.zip?dl=0.

13.5. Creación de bases de datos

El proceso de creación de una base de datos con *ObjectDB* es sumamente sencillo. El primer paso será crear el proyecto Java en el que será utilizada y acto seguido habrá que añadir la librería *objectdb.jar* correspondiente a *ObjectDB/JPA* al proyecto. Dicha librería se encuentra en el subdirectorio *bin* del directorio de instalación de *ObjectDB*.

La primera vez que se ejecute la aplicación se creará automáticamente la base de datos *.odb* en el subdirectorio *db* contenido en el directorio de instalación de *ObjectDB*.

Figura 13.2
Inclusión de la librería *objectdb.jar*.

Actividad propuesta 13.3

Prepara el escenario creando un nuevo proyecto sobre el que llevar a cabo operaciones contra la base de datos y añade la librería de *ObjectDB*.

13.6. El lenguaje de definición de objetos

El lenguaje con el que se definen los objetos y sus miembros en una base de datos orientada a objetos *ObjectDB* emplea un amplio repertorio de marcas que se van añadiendo al código fuente original. A continuación, se mostrará la sintaxis empleada para declarar los diferentes elementos del lenguaje conforme a JPA.

13.6.1. Tipos de datos

Dentro de una base de datos orientada a objetos se almacenan los objetos manejados por la aplicación cuyos datos son de tipo persistente. Los tipos persistente de JPA son los siguientes:

1. Tipos *básicos* (primitivos): *int*, *float*, *byte*, *char*… Puede consultarse el listado completo en el apartado 4.8.
2. Tipos *estructurados*: *array* y *String*. El tipo *array* será persistente siempre que sus elementos sean nulos (*null*) o instancias de tipos persistentes.
3. Tipos *envoltorio*: *Integer*, *Float*, *Double*…
4. Tipos *Math*: *BigDecimal*, *BigInteger*.
5. Tipos para fecha y hora (*Date*, *Time*, *Timestamp* y *Calendar*). Es importante distinguir entre los tipos contenidos en el paquete *java.util* y los contenidos en el paquete *java.sql*. Los contenidos del paquete *java.sql* no necesitan de ningún tipo de marcado para ser tipos persistentes. Sin embargo, los tipos contenidos en el paquete *java.util* necesitan de la marca *@Temporal*. A continuación, se muestran varios ejemplos de declaraciones de tipos fecha y hora:

```
// ejemplos tipos fecha persistentes
java.sql.Date ejemploDate1;
@Temporal(TemporalType.DATE) java.util.Date ejemploDate2;
@Temporal(TemporalType.DATE) java.util.Calendar ejemploDate3;

// ejemplos tipos hora persistentes
java.sql.Time ejemploTime1;
@Temporal(TemporalType.TIME) java.util.Date ejemploTime2;
@Temporal(TemporalType.TIME) java.util.Calendar ejemploTime3;

// ejemplos tipos fecha-hora persistentes
java.sql.Timestamp ejemploDateTime1;
@Temporal(TemporalType.TIMESTAMP) java.util.Date ejemploDateTime2;
@Temporal(TemporalType.TIMESTAMP) java.util.Calendar ejemploDateTime3;
```

6. Tipos *enumerados* (tanto predefinidos como definidos por el programador). De forma predeterminada son manejados a través de los números ordinales. A través de la marca *@Enumeration* se puede cambiar la representación interna para que sean manejados a través de sus nombres. A continuación se muestra un fragmento de código de ejemplo:

```
// declaración del tipo enumerado en una clase
enum TipoFruta { PERA, NARANJA, CEREZA, UVA };

// modificación de las representaciones internas del enumerado
TipoFruta fruta1; // gestionada como EnumType.ORDINAL
@Enumerated(EnumType.ORDINAL) TipoFruta fruta2;
@Enumerated(EnumType.STRING) TipoFruta fruta3;
```

7. Tipos que implementen la interfaz *serializable* (tanto predefinidos como definidos por el programador).
8. Cualquier *colección* o *mapa*, siempre que sus valores (elementos en *colecciones* y claves y valores en *mapas*) sean nulos (*null*) o instancias de tipos persistentes.
9. Cualquier *clase* definida por el usuario y que sea marcada como *Entity*, *Mapped* o *Embeddable*. Las instancias de clases marcada como *Entity* se almacenarán en la base de datos directamente; el resto se pueden incrustar como campos de estas. En el apartado siguiente ser mostrará cómo especificar las clases de estos tipos.

13.6.2. Clases

Para conseguir almacenar la información de los objetos de una determinada clase, será necesario declararla como *clase entidad*. Este tipo de clase se corresponde con el de una clase POJO (*Plain Old Java Object*) y únicamente almacena en la base de datos la información relativa a los campos, obviando sus métodos. Estas clases deben cumplir con:

- ✓ Tener un constructor sin argumentos.
- ✓ Sus campos deben ser privados.
- ✓ Sus propiedades deben ser accesibles mediante métodos *get* y *set* que siguen una convención de nomenclatura estándar (será descrita más adelante).
- ✓ Ser serializables.

Sabías que...

Las instancias de clases *@Embeddable* serán contenidas dentro de objetos persistentes y no requieren un espacio específico de almacenamiento, por lo que ahorran espacio en la base de datos y mejoran su eficiencia. Por el contrario, este tipo de clases no tienen clave primaria, lo cual imposibilita que objetos de este tipo puedan ser compartidos por varias clases entidad.

Cada clase entidad debe ser serializable y ha de declararse con la marca *@Entity* justo antes de la definición de la clase. Las entidades serán representadas en las consultas por sus nombres de entidad y dicho nombre, por defecto, coincidirá con el nombre de la clase, aunque puede ser modificado a otro valor mediante la marca *@Entity(name="NombreEntidad")*:

```
@Entity public class Clase1 implements Serializable {
   ...
}

@Entity(name="nombre") public class Clase2 implements Serializable {
   ...
}
```

Para saber más

Dado que una clase entidad podrá estar formada por campos cuyo tipo sea una clase definida por el usuario, dicha clase deberá ser declarada mediante la marca @Embeddable:

```
@Embeddable public class ClaseComponente {
   ...
}

@Entity public class Entidad implements Serializable {
   ...
   // declaración de campo de tipo ClaseComponente declarado previamente.
}
```

Desde *NetBeans* se pueden crear entidades automáticamente sin necesidad de añadir la marca de forma manual. Para ello es necesario hacer clic derecho sobre el paquete del proyecto, acceder a la opción *New* del menú emergente y hacer clic en la opción *Other* del menú. Una vez se abra el asistente, se ha de marcar la categoría *Persistence*, y en tipos de fichero elegir *Entity Class*. Habrá que desmarcar el *checkbox* que indica *Create Persistence Unit*, y hacer clic en *Finish*.

13.6.3. Campos

Todos los campos de una clase cuyo tipo sea alguno de los especificados en el apartado 13.6.1. serán campos persistentes y, por tanto, serán almacenados en la base de datos sin necesidad de añadir ningún tipo de marca. Sin embargo, si el campo ha sido declarado como *final*, *static* o *transient*, o si ha sido marcado con la etiqueta *@Transient*, no será persistente y su información no se volcará a la base de datos. En el ejemplo siguiente puede verse una declaración de clase entidad con campos no persistentes:

```
@Entity public class EntidadCamposNoPersistentes implements Serializable {
 private final int transient1 = 0;  // no persistente
 private static int transient2; // no persistente
 private transient int transient3; // no persistente
   @Transient private int transient4; // no persistente
   ...
}
```

Como ya se ha indicado en el apartado anterior, una clase entidad no se compone únicamente de campos de tipo primitivo, sino que también puede tener campos de un tipo correspondiente a una clase definida por el programador y que haya sido marcada como *@Embeddable*. Estos campos deberán ser marcados como *@Embedded*. A continuación, se puede ver un ejemplo de uso de estas etiquetas para una relación de asociación entre las clases *Vehículo* y *Persona*:

```
@Embeddable public class Persona {
   private String nombre;
   private String dni;
   ...
}

@Entity public class Vehiculo implements Serializable {
   @Embedded private Persona conductor; // campo tipo Persona
   private String matricula;
   ...
}
```

Recuerda

- ✓ La visibilidad de los campos debe establecerse a *private* para forzar a que sean accedidos mediante propiedades (métodos *get* y *set*).

A) Clave primaria

Para cada entidad es conveniente crear un campo de identificación único, *clave primaria*, que represente de forma unívoca a cada objeto en la base de datos. La clave primaria se declara con la marca *@Id*. Además, es posible indicar cómo se establecerá el valor de dicho campo mediante la marca *@GeneratedValue*. Existen varias alternativas para establecer su valor:

- ✓ Omitir la marca *@GeneratedValue* en un campo declarado como *clave primaria* (*@Id*). Deberá ser el propio programa el que establezca su valor antes de proceder al almacenado en la base de datos.
- ✓ *@GeneratedValue*: si no se indica la estrategia de generación por defecto es *GenerationType.AUTO*.
- ✓ *@GeneratedValue(strategy=GenerationType.AUTO)*: mantiene un generador de números global especial para cada base de datos. El identificador generado es único y nunca se reutiliza aunque se elimine el objeto que contenía dicha clave.
- ✓ *@GeneratedValue(strategy=GenerationType.IDENTITY)*: mantiene un generador de números único para cada jerarquía de tipos.
- ✓ *@GeneratedValue(strategy=GenerationType.SEQUENCE, generator="seq")*: permite utilizar una secuencia, que deberá ser definida previamente, donde se permita establecer el valor inicial de la misma y el tamaño máximo. Un ejemplo de declaración de esta secuencia es *@SequenceGenerator(name="seq", initialValue=1, allocationSize=100)*.

En el siguiente ejemplo se declara una clase entidad con su clave primaria *id*:

```
@Entity public class ClaseX {
   @Id @GeneratedValue(strategy=GenerationType.AUTO)
   private Long id; // persistente y clave principal
   ...
}
```

En el caso de claves primarias compuestas por más de un campo se requiere crear una clase independiente *@Embeddable* que contenga como miembros los campos que compondrán la clave primaria e indicar en la clase entidad que su clave principal es del tipo de clase creada para tal efecto mediante la marca *@EmbeddedId*:

```
@Embeddable Class ClaseClavePrimaria {
    String campo1;
    Date campo2;
}

@Entity public class Producto {
    @EmbeddedId private ClaseClavePrimaria id;
    ...
}
```

Investiga

Busca información sobre la estrategia de generación de claves primarias *GenerationType.TABLE* y describa su funcionamiento.

¿Guarda relación con alguna de las descritas anteriormente?

¿Existe alguna otra alternativa para declarar claves primarias compuestas?

B) Métodos y propiedades

A pesar de que en la base de datos únicamente se almacena la información contenida en los campos de tipo persistente obviándose los métodos que puedan existir, estos toman un papel importante si la entidad usa *propiedades persistentes*. En este caso, la entidad deberá seguir las convenciones de los componentes de *JavaBeans*.

Las *propiedades JavaBeans* utilizan métodos de obtención y establecimiento que toman su identificador del nombre de los campos de la clase entidad. Para cada propiedad persistente llamada *propiedad* se declaran los métodos *getPropiedad* y *setPropiedad*, que permiten recuperar y establecer, respectivamente, el valor del campo. A continuación, se muestra un ejemplo de implementación de propiedades para una clase compuesta de dos campos:

```
@Embeddable public class Persona {
    private String nombre;
    private String dni;

    public String getNombre() { return nombre; }
    public void setNombre(String nombre) { this.nombre = nombre; }
    public String getDni() { return dni; }
    public void setDni(String dni) { this.dni = dni; }
}
```


Para saber más

Una forma muy cómoda de generar los métodos *get* y *set* de forma automática desde *NetBeans* es colocando el puntero del ratón sobre el código de la clase y pulsando la combinación de teclas *Alt + Insert*. Aparecerá un menú en el que se puede hacer clic en la opción *Getter and Setter…* y a partir de ahí seleccionar los campos para los cuales se quieren generar.

13.6.4. Constructores

Para cada entidad es fundamental que exista un constructor vacío y sin argumentos (constructor por defecto) cuya visibilidad sea *public* o *protected* y que no debe sobrescribirse. Además de este constructor la clase podrá tener tantos constructores parametrizados como sea necesario:

```
@Embeddable public class Persona {
    private String nombre;
    private String dni;

    Persona (String nombre, String dni){
        this.nombre = nombre;
        this.dni = dni;
    }

    ... // métodos get y set
}
```

13.6.5. Herencia

La misma relación de herencia que puede declararse en el lenguaje orientado a objetos se puede declarar entre las clases entidad que serán almacenadas en la base de datos. Cuando se almacenan los campos de una clase entidad también se almacenarán los campos persistentes de las clases ancestras.

Las relaciones de herencia pueden implementarse tanto en clases entidad (*@Entity*) como en clases componente (*@Embeddable*).

13.7. Mecanismos de consulta

La *Java Persistence API* proporciona dos alternativas para consultar entidades desde una base de datos orientada a objetos: *Java Persistence Query Language* (JPQL) y *API Criteria*. Ambas están

estrechamente relacionadas y han sido diseñadas para permitir operaciones similares en sus consultas. Desde un punto de vista comparativo, ambas alternativas presentan ventajas e inconvenientes:

- JPQL es un lenguaje simple de consultas para bases de datos orientadas a objetos, basado en cadenas e independiente de la plataforma. Una ventaja de esta alternativa es que las consultas suelen ser más concisas y legibles que las consultas de *API Criteria*. Otra ventaja es que es más sencilla de comprender para los desarrolladores familiarizados con SQL debido a su similitud. Sin embargo, también presenta algún inconveniente: las consultas no son seguras y requieren de una conversión cuando se recupera el resultado de la consulta desde el administrador de entidades, lo cual puede provocar errores de conversión de tipos que pueden no ser detectados en tiempo de compilación.
- *API Criteria* se utiliza para crear consultas de tipos seguros utilizando las API de lenguaje de programación Java para consultar las entidades y sus relaciones. Las consultas de la *API Criteria* permiten definir la consulta en el nivel de negocios de la aplicación. Además, son seguras y no requieren conversión. Otra ventaja es que no es necesario que los desarrolladores aprendan la sintaxis de otro lenguaje de consulta como sucede con JPQL.

Actividad propuesta 13.4

Realiza un cuadro resumen comparativo de los dos mecanismos de consulta mostrados.

13.7.1. El lenguaje de consultas: sintaxis, expresiones, operadores

Antes de mostrarse los mecanismos empleados para realizar las típicas operaciones *CRUD*, es imprescindible describir la sintaxis de las diferentes *query* JPQL en ellas empleadas, así como las expresiones y operadores que se pueden incluir.

Como base para todas las *query* se parte de las siguientes consideraciones generales:

- ✓ Una *query* no es sensible a mayúsculas o minúsculas excepto para el nombre de las entidades, atributos y las relaciones.
- ✓ Los datos cadena de texto estarán formados por caracteres Unicode encerrados entre comillas simples. El carácter literal comilla simple (') se representará mediante la doble comilla simple ('').
- ✓ Los datos numéricos seguirán la sintaxis de Java y SQL. Pueden utilizarse tipos numéricos exactos y en punto flotante.
- ✓ Los datos *booleanos* se representan mediante *TRUE* o *FALSE*.
- ✓ Los datos enumerados seguirán la sintaxis de Java (es decir, irán precedidos por la clase en la que fueron declarados).
- ✓ Se podrán emplear parámetros posicionales y por nombre.

A) SELECT

Este tipo de *query* está destinada a consultas y podrá devolver entidades u otro tipo de datos (información de los campos de una entidad, cálculos realizados sobre los datos recuperados…). Su sintaxis general de uso es:

```
SELECT [DISTINCT] ...
FROM ... [AS] ...
[WHERE ...]
[GROUP BY ... [HAVING ...]]
[ORDER BY ...]
```

siendo los elementos declarados entre corchetes ([]) opcionales. En el cuadro 13.1 puede verse una descripción de las palabras reservadas empleadas en las *query* de consulta.

Cuadro 13.1
Palabras reservadas empleadas en consultas

SELECT	Determina las entidades y campos a ser recuperados. Y los valores resultantes de utilizar funciones agregadas (cuadro 13.2) y funciones aritméticas, para manejo de cadenas y para fecha-hora (cuadro 13.4).
FROM	Declara el identificador de la entidad de la que se quiere obtener la información (al menos una). Si son más van separadas por comas ",".
WHERE	Es empleada para filtrar los resultados devueltos por la *query* mediante el empleo de operadores (cuadro 13.3) y funciones aritméticas, para manejo de cadenas y para fecha-hora (cuadro 13.4).
GROUP BY	Es empleada para agrupar los resultados.
HAVING	Similar a WHERE pero para filtrar sobre los grupos (solo se utiliza cuando se incluye GROUP BY). Podrán utilizarse las funciones agregadas (cuadro 13.2) y funciones aritméticas, para manejo de cadenas y para fecha-hora (cuadro 13.4).
ORDER BY	Permite ordenar los resultados devueltos por la *query*. Por defecto ordena de forma ascendente. Es posible indicar además el criterio de ordenación de forma explícita mediante ASC o DESC.

La palabra reservada DISTINCT en la sentencia SELECT elimina de los resultados aquellos que estén duplicados.

Cuadro 13.2
Funciones agregadas

AVG	Retorna la media de los valores de los campos.
COUNT	Retorna el número total de resultados.
MAX	Devuelve el valor más alto del conjunto de resultados.

[…/…]

Cuadro 13.2 (continuación)

MIN	Devuelve el valor más bajo del conjunto de resultados.
SUM	Devuelve la suma del conjunto de resultados.

Los operadores que se pueden utilizar son descritos en el cuadro 13.3 y aparecen mostrados de mayor a menor orden de precedencia.

Cuadro 13.3
Operadores

Navegación y acceso a campos	. (punto)
Aritméticos	+ y - (unarios), * (multiplicación), / (división), + (suma), - (resta)
Comparadores	=, <, <=,>,>=, <>, [NOT] BETWEEN, [NOT] LIKE, [NOT] IN, IS [NOT] NULL, IS [NOT] EMPTY, [NOT] MEMBER [OF]
Lógicos	NOT, AND, OR

Cuadro 13.4
Funciones aritméticas, de manejo de cadenas y para fecha-hora

CONCAT(String, String)	Une dos cadenas.
LENGTH(String)	Devuelve la longitud de la cadena.
LOCATE(String, String [, pos])	Devuelve la posición de una cadena en otra. Es posible indicar la posición desde la que buscar. En un *String* la primera posición es la número 1.
SUBSTRING(String, pos, num)	Devuelve una subcadena de cadena, con los *num* caracteres existentes desde la posición *pos*.
TRIM([[LEADING|TRAILING|BOTH] char) FROM] (String)	Elimina el carácter indicado del principio, final o ambos. Si no se indica el carácter eliminará los espacios en blanco.
LOWER(String)	Convierte a minúsculas la cadena.
UPPER(String)	Convierte a mayúsculas la cadena.
ABS(num)	Retorna el valor absoluto del número.
MOD(n1, n2)	Devuelve el resto de n1/n2.
SQRT(double)	Devuelve la raíz cuadrada del número.
SIZE(Collection)	Retorna el número de elementos de la colección.
CURRENT_DATE	Devuelve la fecha actual.
CURRENT_TIME	Devuelve la hora actual.
CURRENT_TIMESTAMP	Devuelve la fecha y hora actual.

A continuación, se muestran ejemplos de consultas realizadas a partir de una entidad *Producto* cuyos campos son *identificador* (*id*), *nombre*, *descripción*, *precio* y *país* de procedencia:

SELECT p FROM Producto p	Devolverá todas las entidades Producto.
SELECT p FROM Producto p WHERE p.precio> 30	Devolverá las entidades Producto cuyo precio sea mayor a 30.
SELECT p FROM Producto p WHERE p.precio =?1	Devolverá las entidades Producto cuyo precio sea igual al parámetro posicional *1*.
SELECT p.nombre, p.id FROM Producto p WHERE p.precio BETWEEN :precio1 AND :precio2 ORDER BY p.nombre DESC	Devolverá los nombres e identificadores (id) de productos cuyo precio esté comprendido entre dos precios pasados como parámetros por nombre (*precio1* y *precio2*). El resultado estará ordenado descendentemente por nombre de producto.
SELECT p.pais, MAX(p.precio) FROM Producto p GROUP BY p.pais HAVING p.pais IN ('FRANCIA', 'PORTUGAL', 'ITALIA')	Devolverá el país y el precio máximo de los productos de cada país, de aquellos países contenidos en el conjunto especificado (FRANCIA, PORTUGAL e ITALIA).

Actividades propuestas

13.5. En una sentencia SELECT es posible relacionar información de 2 o más entidades mediante los operadores JOIN. Investiga los diferentes tipos de operador JOIN y propón ejemplos de uso de estos.

13.6. ¿Es posible incluir HAVING en una query que no emplee GROUP BY? Justifica tu respuesta.

Sabías que...

El operador *LIKE* puede hacer uso de algunos caracteres comodín:

- ✓ El carácter porcentaje (%), que coincide con cualquier cadena de cero o más caracteres.
- ✓ El carácter guion bajo (_), que coincide con un único carácter.

B) UPDATE

Este tipo de *query* está destinada a la actualización de datos de entidades. La sintaxis de uso de la sentencia UPDATE es:

```
UPDATE ... [AS] ...
SET ... [WHERE ...]
```

En el cuadro 13.5 puede verse una descripción de las palabras reservadas empleadas en las *query* de actualización.

CUADRO 13.5
Palabras reservadas empleadas en consultas

UPDATE	Determina el tipo de entidad a modificar.
SET	Establece cómo realizar la modificación. Se podrán utilizar las funciones aritméticas para manejo de cadenas y para fecha-hora descritas anteriormente para SELECT, WHERE y HAVING.
WHERE	Es empleada para especificar las entidades sobre las que realizar el cambio mediante el empleo de *operadores* y *funciones*.

A continuación, se muestran ejemplos realizados a partir de la entidad *Producto*, anteriormente propuesta para los ejemplos de SELECT:

`UPDATE Producto AS p` `SET p.pais = 'Venezuela'`	Establece el país de origen de todos los productos a 'Venezuela'.

`UPDATE Producto`
`SET pais = 'Espana'`
`WHERE descripcion LIKE '%Espana%'`
	Establece el país de origen a 'Espana' en todos los productos que contengan en su descripción el literal 'Espana'.

C) DELETE

La *query* DELETE está destinada al borrado de la información correspondiente a las entidades de la base de datos. Su sintaxis de uso es:

```
DELETE FROM ... [AS] ...
[WHERE ...]
```

En el cuadro 13.6 puede verse una descripción de las palabras reservadas empleadas en las *query* de borrado.

CUADRO 13.6
Palabras reservadas empleadas en borrados de información

DELETE FROM	Determina el tipo de entidad a eliminar.
WHERE	Es empleada para especificar las entidades concretas a eliminar mediante *operadores* y *funciones*.

A continuación, se muestran algunos ejemplos realizados a partir de la entidad *Producto*:

`DELETE FROM Producto`	Elimina todas las entidades Producto.

`DELETE FROM Producto AS p`
`WHERE p.nombre = 'naranja' AND`
`p.pais = 'Espana'`
	Elimina todas las entidades Producto cuyo nombre sea 'naranja' y país de procedencia sea 'Espana'.

PARA SABER MÁS

No es objetivo de este libro abordar la totalidad de los elementos que componen el lenguaje de consultas JPQL, por lo que únicamente se han descrito los elementos principales. En la URL https://docs.oracle.com/html/E13946_04/ejb3_langref.html podrás obtener una extensa guía con información sobre JPQL.

13.7.2. Establecimiento de la conexión

El primer paso para el establecimiento de la conexión es crear el objeto que represente a la base de datos (*EntityManagerFactory*). La ruta de la base de datos podrá establecerse:

- ✓ Utilizando la variable *$objectdb* que toma su valor a partir del *path* en el que se encuentra la librería *objectdb.jar*.
- ✓ Utilizando directamente el *path* de la base de datos ("*C:\ ...*").
- ✓ Utilizando una URL en caso de que la base de datos sea accesible a través de un servidor *ObjectDB*. Si se opta por esta alternativa, es posible indicar parámetros de conexión en la URL ("*objectdb://localhost/BBDD.odb;user=USUARIO;password=PASS*").

A partir de dicho objeto se deberá construir el gestor de entidades (*EntityManager*) que representará la conexión a dicha base de datos. Una vez la conexión haya sido establecida se podrán realizar las operaciones *CRUD* (*Create*, *Read*, *Update*, *Delete*) sobre la base de datos a través de las *query* descritas en el apartado anterior. Cuando ya no se vayan a realizar más operaciones sobre la base de datos, será necesario cerrar la conexión. A continuación, se muestra un ejemplo de establecimiento y cierre de conexión a una BD:

```
EntityManagerFactory emf;
emf = Persistence.createEntityManagerFactory("$objectdb/db/BBDD.odb");
EntityManager em;
em = emf.createEntityManager();
 ... // operaciones CRUD
em.close();
emf.close();
```

13.7.3. Inserción, recuperación, modificación y borrado de información

A continuación, se describen los pasos para llevar a cabo las operaciones *CRUD* contra un sistema gestor de bases de datos orientadas a objetos. En todo momento se parte de la base de que se ha realizado la conexión hacia la base de datos y de que esta se cerrará una vez sea realizada la operación. Las operaciones que impliquen modificaciones en los contenidos de la base de datos tendrán que realizarse siempre dentro de una transacción. Para las operaciones JPQL deberá atenerse a lo descrito en el apartado 13.7.1.

A) Inserción (Create)

Esta operación permitirá crear entidades en la base de datos. Los pasos que se han de seguir serán: iniciar la transacción (*begin*), crear los objetos de *clase entidad* a insertar, ejecutar el método *persist* del gestor de entidades para cada objeto creado y, por último, finalizar la transacción (*commit*). A continuación, se muestra la secuencia anteriormente descrita:

```
em.getTransaction().begin();
ClaseEntidad objeto = new ClaseEntidad (arg1, arg2, ..., argN);
em.persist(objeto);
em.getTransaction().commit();
```

```
// Almacenando datos (Create)
em.getTransaction().begin();
Trabajador t1 = new Trabajador("Mario", "77777777X", 888);
Trabajador t2 = new Trabajador("Ana", "55555555J", 111);
Persona p1 = new Persona("David", "99999999H");
Persona p2 = new Persona("Gabriel", "33333333K");
Calendar f1 = Calendar.getInstance(); f1.set(2000, 10, 20);
Coche c1 = new Coche(t1, "GGG-3333", Coche.EnergiaPropulsion.DIESEL, f1);
Calendar f2 = Calendar.getInstance(); f2.set(2017, 0, 13);
Coche c2 = new Coche(p1, "SG-1111-C", Coche.EnergiaPropulsion.HIBRIDO, f2);
Calendar f3 = Calendar.getInstance(); f3.set(2015, 1, 28);
Vehiculo v1 = new Vehiculo(t2, "CCC-????", f3);
Calendar f4 = Calendar.getInstance(); f4.set(2019, 0, 31);
Vehiculo v2 = new Vehiculo(p2, "FWW-6666", f4);
em.persist(c1); em.persist(c2); em.persist(v1); em.persist(v2);
em.getTransaction().commit();
```

Figura 13.3
Ejemplo de inserción.

Actividades propuestas

13.7. ¿Qué tipo de *query* JPQL se utiliza para realizar la inserción de objetos en una BDOO?

13.8. Escribe un programa que realice la inserción de 5 objetos de la clase producto empleada en el apartado 13.7.1 en una base de datos denominada *Productos*.

B) Recuperación (Read)

La operación de recuperación se corresponde con una típica consulta de datos. Existen varias alternativas para recuperar información:

1. *Recuperación mediante identificador*: cada objeto puede ser identificado unívocamente, y por tanto recuperado, por el nombre de su clase entidad y el valor del campo clave primaria. Para ello se emplea el método *find* del gestor de entidades que recibe como parámetro la clase entidad en la que buscar y el valor que identifica por la clave primaria al objeto. El resultado de este método es *null* en caso de no recuperarse el objeto o un objeto de la clase indicada, por lo que no es necesario hacer un *casting*. En el siguiente ejemplo se muestra cómo realizar este tipo de operación.

```
ClaseEntidad objeto = em.find(ClaseEntidad.class, valorClavePrincipal);
```

```
// Recuperando un objeto de la BD a partir de su ID (Read)
Coche coche = em.find(Coche.class,1);
System.out.println("Coche: " + coche.getMatricula());
```

Figura 13.4
Recuperación mediante identificador.

2. *Recuperación mediante query*: se trata del método más flexible y completo. Los resultados se recuperarán mediante los métodos siguientes:
   - *getSingleResult* cuando se espera obtener un resultado único (el mayor valor, la media de un conjunto de valores, un valor concreto…).
   - *getResultList* cuando se espera que se retorne un listado de resultados. Este resultado deberá manejarse como un listado.

La *query* puede declararse como *estática* o *dinámica*.

a) Una *query estática* permite mantener más ordenado el código pues es declarada fuera de la entidad, separando de esta manera las cadenas de consulta JPQL del código Java. Es más eficiente que una *query dinámica*, pero también es menos flexible. Este tipo de *query* es declarada mediante anotaciones:

```
@NamedQuery(name="queryEstática", query="SELECT ...")
@Entity ...
```

Podrán definirse tantas como sea necesario, pero en caso de ser más de una deberán declararse dentro de una marca *@NamedQueries* y siempre con distinto nombre:

```
@NamedQueries({
   @NamedQuery(name="queryEstática1", query="SELECT ..."),
   @NamedQuery(name="queryEstática2", query="SELECT ...")
})
@Entity ...
```

La cadena de la consulta JPQL será recuperada por el método *createNamedQuery* al que se le pasa como argumento el nombre con el que fue declarada previamente. Este método está sobrecargado, siendo su sintaxis la mostrada a continuación:

```
Query consulta = em.createNamedQuery("queryEstática");
```

```
TypedQuery<Clase> consulta = em.createNamedQuery("queryEstática", ClaseEntidad.class);
```

```
@NamedQuery(name="queryEstática", query="SELECT v FROM Vehiculo v ")
@Entity public class Vehiculo implements Serializable {
```

```
// Recuperando datos con query estática (Read)
Query query1 = em.createNamedQuery("queryEstática");
List<Vehiculo> resultados1 = query1.getResultList();
for (Vehiculo v : resultados1) {
    System.out.println("Matrículas de vehículos: " + v.getMatricula());
}
```

Figura 13.5
Recuperación mediante query estática.

b) Una *query dinámica* es creada en tiempo de ejecución por la aplicación. Por tanto, es útil cuando la estructura de la consulta depende de las entradas del usuario u otras condiciones. La *query* será creada mediante el método *createQuery*. Si como argumento únicamente se pasa la consulta JPQL, el resultado no pertenecerá a ningún tipo conocido. Sin embargo, es posible pasar como argumento también la clase de los objetos a recuperar.

```
Query consulta = em.createQuery("SELECT ...");
```

```
TypedQuery<Clase> consulta = em.createQuery("SELECT ...", ClaseEntidad.class);
```

```
// Recuperando datos con query dinámica (Read)
TypedQuery query2a = em.createQuery("SELECT COUNT(c) FROM Coche c",Coche.class);
Query query2b = em.createQuery("SELECT COUNT(v) FROM Vehiculo v");
System.out.println("Coches en total: " + query2a.getSingleResult());
System.out.println("Vehículos en total: " + query2b.getSingleResult());
```

Figura 13.6
Recuperación mediante query dinámica.

Tal como se indicó anteriormente, es posible añadir parámetros a la consulta mediante el método *setParameter* para hacer que la *query* sea dinámica.

– Cada *parámetro posicional* se identificará dentro del literal de la *query* por **?n**, siendo **n** el número identificador de parámetro. Ejemplo:

```
// Recuperando datos con query dinámica y parámetro posicional (Read)
Query query4 = em.createQuery("SELECT v.matricula FROM Vehiculo v "
        + "WHERE id = ?1");
query4.setParameter(1, 4);
System.out.println("Matrícula de vehículo id=4: " + query4.getSingleResult());
```

Figura 13.7
Ejemplo recuperación por query dinámica y parámetro posicional.

– Cada *parámetro por nombre* se identifica por llevar como prefijo el carácter dos puntos (:). Ejemplo:

```
// Recuperando datos con query dinámica y parámetro por nombre (Read)
Query query3 = em.createQuery("SELECT v FROM Vehiculo v "
        + "WHERE matricula LIKE :mat");
query3.setParameter("mat", "SG%");
List<Vehiculo> resultados2 = query3.getResultList();
for (Vehiculo v : resultados2) {
    System.out.println("Matrículas de Segovia: " + v.getMatricula());
}
```

Figura 13.8
Ejemplo recuperación por query dinámica y parámetro por nombre.

PARA SABER MÁS

Además de los métodos *createNamedQuery* y *createQuery* vistos en este apartado, también se dispone del método *createNativeQuery(String sql)* , el cual permite lanzar *querys* nativas de SQL sobre el SGBD.

Actividades propuestas

13.9. ¿Es posible realizar la consulta dentro de una transacción? Justifica tu respuesta.

13.10. Implementa un programa que recupere los datos de los 5 objetos de la clase producto insertados en la actividad propuesta 13.8 (realiza la recuperación mediante identificador y mediante *query*).

C) *Modificación* (Update)

La modificación puede llevarse a cabo también de dos formas bien diferenciadas. Independientemente del método escogido, será necesario realizar los cambios dentro de una transacción:

1. *Modificación mediante identificador*: este proceso consiste en llevar a cabo la recuperación de la entidad a modificar (método *find*) y la posterior modificación del contenido recuperado mediante las propiedades del objeto. Al tratarse de una actualización, y no de una inserción, no debe invocarse el método *persist*. Ejemplo:

```
ClaseEntidad objeto = em.find(ClaseEntidad.class, valorClavePrincipal);
em.getTransaction().begin();
objeto.setArg1(valor);
em.getTransaction().commit();
```

```
// Modificando un objeto de la BD a partir de su ID (Update)
Vehiculo vehiculo = em.find(Vehiculo.class, 1);
em.getTransaction().begin();
vehiculo.setMatricula("8161-LLL");
em.getTransaction().commit();
```

Figura 13.9
Modificación mediante identificador.

2. *Modificación mediante query*: al igual que en las consultas se podrá declarar la *query* como *estática* o *dinámica*. Ejemplo:

```
em.getTransaction().begin();
Query consulta = em.createQuery("UPDATE ...");
int numeroObjetosModificados = consulta.executeUpdate();
em.getTransaction().commit();
```

```
// Modificando datos con query dinámica (Update)
em.getTransaction().begin();
Query consulta = em.createQuery("UPDATE Vehiculo "
        + "SET matricula = '8161-LLL' "
        + "WHERE id = 1 ");
int numeroObjetosModificados = consulta.executeUpdate();
em.getTransaction().commit();
```

Figura 13.10
Modificación mediante query.

Como resultado de invocar el método *executeUpdate*, se obtendrá el número de objetos que hayan sido modificados por la instrucción.

Actividad propuesta 13.11

Implementa un programa que modifique los datos de algunos de los objetos de la clase producto insertados en la actividad propuesta 13.8 (realiza la modificación mediante identificador y mediante *query*).

D) Borrado (Delete)

El borrado se realizará de forma similar a la modificación:

1. *Borrado mediante identificador*: este proceso consiste en llevar a cabo la recuperación de la entidad (método *find*) y la posterior eliminación del objeto entidad (*remove*). A continuación, se muestra un ejemplo de borrado mediante identificador:

```
ClaseEntidad objeto = em.find(ClaseEntidad.class, valorClavePrincipal);
em.getTransaction().begin();
em.remove(objeto);
em.getTransaction().commit();
```

```
// Eliminando un objeto de la BD a partir de su ID (Delete)
Vehiculo vehiculo = em.find(Vehiculo.class, 3);
em.getTransaction().begin();
em.remove(vehiculo);
em.getTransaction().commit();
```

Figura 13.11
Borrado mediante identificador.

2. *Borrado mediante* query: se realiza exactamente igual que la actualización, pero sustituyendo la *query UPDATE* por una *DELETE*.

Actividad propuesta 13.12

Implementa un programa que elimine los datos del objeto de la clase producto insertado en la actividad propuesta 13.8 con el identificador más bajo (realice la modificación mediante identificador y mediante *query*).

13.8. Explorador de objetos

Además del propio sistema gestor de base de datos, *ObjectDB* incluye la herramienta *ObjectDB Explorer*. Se trata de una aplicación con interfaz gráfico que permite administrar de forma cómoda las bases de datos, ofreciendo la posibilidad de editar su contenido, así como de ejecutar código JPQL:

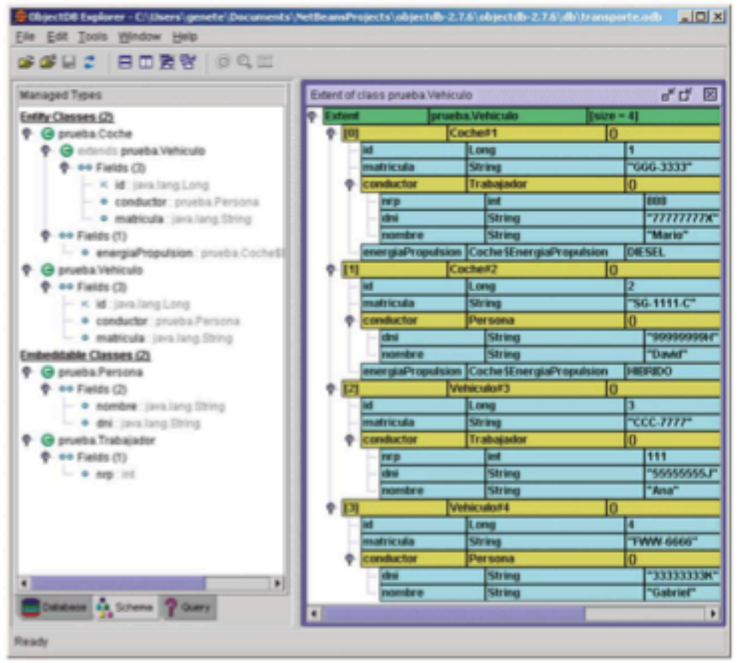


Figura 13.12 Explorador de objetos de *ObjectDB*.

Resumen

- El empleo de ficheros para el almacenado de grandes cantidades de información dificulta su manejo y es poco eficiente.
- Una base de datos orientada a objetos permite hacer persistente la información de los objetos manteniendo las relaciones entre ellos existentes según el paradigma de orientación a objetos.
- JPA es una API para Java que proporciona mecanismos de gestión de la persistencia de objetos en bases de datos orientadas a objetos.
- Cada clase declarada como clase entidad es un POJO y, por tanto, debe cumplir con determinados requisitos (implementar interfaz serializable, constructor sin argumentos, campos privados y propiedades públicas para el acceso a los campos).
- Se recomienda que cada clase entidad declare uno de sus campos como clave primaria.
- Las consultas contra la base de datos pueden realizarse tanto con JPQL (similar a SQL) como con la *API Criteria*. Antes de realizarse una consulta debe abrirse la conexión contra la base de datos. Una vez efectuada debe cerrarse antes de finalizar el programa.

Ejercicios propuestos

1. Investiga la sintaxis de uso de la *API Criteria* y propón algún ejemplo.
2. ¿Es posible mezclar parámetros por nombre y parámetros posicionales en una *query*?
3. Investiga sobre el marcado de clases con la etiqueta *@Mapped*. ¿Qué significado tiene?
4. ¿Para qué se utilizan los *scripts enhancer.bat* y *enhancer.sh*?
5. Indica qué significado tiene marcar un campo persistente con alguna de las siguientes marcas: *@OneToOne*, *@ManyToOne*, *@OneToMany*, *@ManyToMany*, *@Basic*. Propón ejemplos ilustrativos.

Supuestos prácticos

1. Se necesita implementar un programa que administre la información de los vehículos y personas que los conducen cuando entran en un recinto privado. De los vehículos se desea almacenar su identificador (número entero), la matrícula (cadena de caracteres), la fecha de matriculación (fecha) y la persona que lo conduce (clase persona). De los vehículos que sean coches, además, se desea almacenar el tipo de energía que lo propulsa (enumerado: GASOLINA, DIÉSEL, ELÉCTRICO, HÍBRIDO). De las personas que conducen los vehículos se desea almacenar su nombre (cadena de caracteres) y su DNI (cadena de caracteres). De aquellas personas que además sean trabajadores se desea almacenar su número de registro personal (número entero). Implementa las clases entidad que permitan manejar y volcar esta información a una base de datos denominada control, así como un programa que añada los datos de 4 vehículos (2 de ellos coches) y 4 conductores (2 de ellos trabajadores). Utiliza el explorador *ObjectDB Explorer* para comprobar que se han introducido correctamente.
2. Sobre el supuesto práctico anterior, añade al programa principal un menú que ofrezca al usuario la posibilidad de dar de alta nuevos vehículos y personas, consultar los datos de los vehículos almacenados en el sistema, modificar los mismos datos y eliminar los datos de determinados vehículos, incluidos coches, a partir de su identificador.
3. Crea un programa que permita migrar los datos de los trabajadores de una empresa contenidos en un fichero de texto a una base de datos *ObjectDB*. Los campos contenidos en el fichero de texto vienen separados por el carácter # y son los siguientes: DNI, Apellido1, Apellido2, Nombre, FechaNacimiento, FechaIncorporación, Puesto. Selecciona el tipo de dato persistente que mejor se adapte a la información a almacenar.
4. A partir de la base de datos del supuesto práctico anterior, escribe un programa que permita recuperar los datos de los trabajadores según determinados parámetros que serán introducidos por teclado conforme a la opción elegida en un menú de opciones. El menú constará de 3 opciones:

- "Filtrar mayores de edad": Devolverá únicamente los datos de los trabajadores de más de 18 años a fecha actual.
- "Filtrar por DNI igual a…": Devolverá únicamente los datos del trabajador cuyo DNI coincida con el leído.
- "Filtrar por antigüedad mayor a…": Devolverá los datos de los trabajadores cuya antigüedad en el puesto sea superior a la leída por teclado.

Emplea *@NamedQuery* para una de ellas y *querys* dinámicas con parámetros por nombre para las otras dos.

5. Modifica el programa del supuesto práctico anterior para que añada la opción "Alta trabajador" al menú y permita leer los datos desde teclado para almacenarlos en la base de datos.

Actividades de autoevaluación

1. ¿Cuál de las siguientes no es una declaración de tipo persistente?
   - a) *java.sql.Date fecha.*
   - b) *@Temporal(TemporalType.DATE) java.util.Date fecha.*
   - c) *@Temporal(TemporalType.DATE) java.util.Calendar fecha.*
   - d) Todas las opciones anteriores son declaraciones de tipo persistente.
2. ¿Cuál de los siguientes no es un requisito de una clase entidad?
   - a) Tener un constructor parametrizado.
   - b) Sus campos deben ser privados.
   - c) Sus propiedades deben ser accesibles mediante métodos *get* y *set* que siguen una convención de nomenclatura estándar.
   - d) Ser serializable.
3. ¿Cómo debe marcarse el campo de una clase entidad cuyo tipo sea una clase definida por el programador y del que no se desea que sea almacenado como clase entidad?
   - a) *@Embedded.*
   - b) *@Embeddable.*
   - c) *@Serializable.*
   - d) *@Entity.*
4. De las siguientes estrategias de generación de clave primaria, ¿cuál es equivalente a *@GeneratedValue*?
   - a) Omitir la marca es la equivalencia.
   - b) *@GeneratedValue(strategy=GenerationType.IDENTITY).*
   - c) *@GeneratedValue(strategy=GenerationType.AUTO).*
   - d) *@GeneratedValue(strategy=GenerationType.SEQUENCE, generator="seq").*

5. ¿En qué parte de una sentencia SELECT se pueden emplear las funciones agregadas?
   - a) *SELECT.*
   - b) *FROM.*
   - c) *AS.*
   - d) Todas las opciones son correctas.
6. ¿Cuál de los siguientes métodos propaga los cambios a la base de datos?
   - a) *persist.*
   - b) *begin.*
   - c) *rollback.*
   - d) *commit.*
7. ¿Cuál de las siguientes *query* utiliza parámetros posicionales de forma correcta?
   - a) *em.createQuery("SELECT c FROM Compra c WHERE p.precio = *1").*
   - b) *em.createQuery("SELECT c FROM Compra c WHERE p.precio =?1").*
   - c) *em.createQuery("SELECT c FROM Compra c WHERE p.precio = '*1'").*
   - d) *em.createQuery("SELECT c FROM Compra c WHERE p.precio = '?1'").*
8. ¿Qué operación del conjunto de operaciones CRUD no necesita de un *commit*?
   - a) C (*Create*).
   - b) R (*Read*).
   - c) U (*Update*).
   - d) D (*Delete*).
9. ¿Cuál de las siguientes no es una opción válida para establecer la ruta de la base de datos?
   - a) Utilizar la variable *$objectdb*.
   - b) Utilizar directamente el *path* de la base de datos ("C:\ …").
   - c) Utilizar una URL.
   - d) Todas las opciones anteriores son válidas para establecer la ruta de la base de datos.
10. ¿Qué métodos permiten establecer los cambios que se ejecutan como una transacción?
    - a) *em.getTransaction().begin();* y *em.getTransaction().end().*
    - b) *em.getTransaction().start();* y *em.getTransaction().end().*
    - c) *em.getTransaction().begin();* y *em.getTransaction().commit().*
    - d) *em.getTransaction().start();* y *em.getTransaction().commit().*

SOLUCIONES:

1.	a	b	c	**d**	5.	**a**	b	c	d						
2.	**a**	b	c	d	6.	a	b	c	**d**	9.	a	b	c	**d**	
3.	**a**	b	c	d	7.	a	**b**	c	d	10.	a	b	c	**d**	
4.	a	b	**c**	d	8.	a	**b**	c	d						

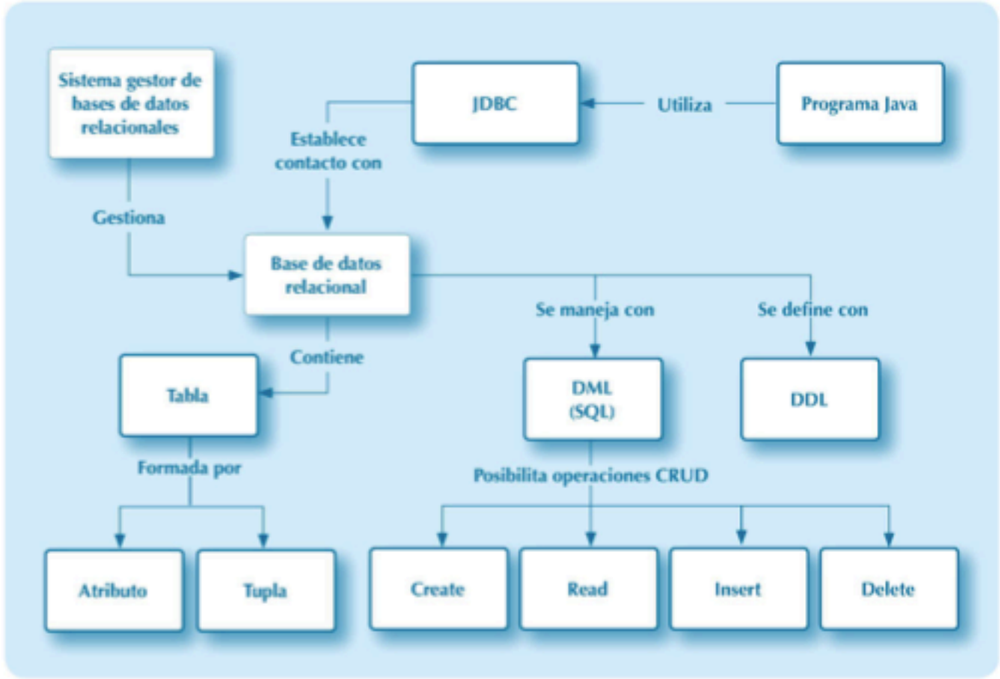
14

Gestión de bases de datos relacionales

Objetivos

- ✓ Instalar un sistema gestor de bases de datos relacionales.
- ✓ Crear bases de datos relacionales.
- ✓ Aprender a obtener los *drivers* de conexión y a establecer conexiones a las bases de datos.
- ✓ Aplicar el Lenguaje de Definición de Datos y el Lenguaje de Manipulación de Datos (SQL) empleados sobre bases de datos relacionales.
- ✓ Navegar por los resultados obtenidos de una *query* recuperando la información y metainformación deseada.
- ✓ Conocer y utilizar asistentes para el manejo de bases de datos relacionales.

Mapa conceptual



Glosario

Autocommit. Mecanismo que permite hacer persistentes los cambios en una base de datos tan pronto como son realizados y sin necesidad de ser confirmados.

DDL. Sigla de *Data Definition Language* ("lenguaje de definición de datos"). Es el conjunto de sentencias proporcionado por el sistema de gestión de base de datos que permite definir las estructuras que almacenarán los datos, así como los procedimientos o funciones que permitan consultarlos.

DML. Sigla de *Data Manipulation Language* ("lenguaje de manipulación de datos"). Es el conjunto de sentencias destinadas a la ejecución de las operaciones CRUD sobre los datos contenidos en las bases de datos del sistema gestor de bases de datos. El más conocido de estos lenguajes es SQL.

Driver. Controlador software que permite la interacción entre dos elementos.

JDBC. Sigla de *Java Database Connectivity*. Es una API utilizada a modo de *driver* que permite a una aplicación Java interactuar con una base de datos.

Modelo relacional. Modelo organizativo de información para bases de datos que almacena los datos en tablas compuestas por campos (columnas o atributos) y tuplas (filas o registros), basándose en la lógica de predicados y en la teoría de conjuntos.

Transacción. Conjunto de instrucciones que deben llevarse a cabo de forma atómica.

Tupla. Conocido también como registro o simplemente fila, es el conjunto de datos existente en una tabla que representa la información de un determinado objeto o entidad.

14.1. Introducción

Habiéndose comprendido la importancia de utilizar bases de datos como alternativa a los tradicionales sistemas de ficheros y después de haber tomado contacto con un sistema de gestión de base de datos orientado a objetos, la gestión de sistemas de bases de datos relacionales no supone mayor problema que el de la adaptación a las diferencias existentes entre ambos SGBD.

En este capítulo se trabajará sobre SGBD relacionales, llevando a cabo su instalación, la creación de BD, la conexión desde diferentes programas y la ejecución de consultas sin entrar en demasiados detalles acerca del lenguaje de consultas empleado.

14.2. Base de datos relacional

Una *base de datos relacional* (*BDR*) es un sistema organizado de datos que representa la información conforme al modelo relacional y que garantiza la integridad referencial. Está compuesta de un conjunto de tablas en las que se almacena la información de cada una de las entidades y las relaciones existentes entre ellas.

Al igual que en las bases de datos orientadas a objetos existe el concepto *clave primaria*, que en este caso se corresponde con un atributo o atributos (columna o columnas), que permite identificar de forma unívoca cada *tupla*.

Este tipo de bases de datos son ampliamente utilizadas y existe un gran número de alternativas tanto libres como propietarias. Para la redacción de este contenido se ha optado por una alternativa libre, concretamente *MariaDB* que es un derivado libre de *MySQL*.

Recurso web www

Desde la URL https://downloads.mariadb.org/ podrás realizar la descarga del SGBD *MariaDB*. Desde la URL https://www.mysql.com/downloads/ podrás realizar la descarga del SGBD *MySQL*.

Actividades propuestas

14.1. ¿Qué es un *ORM*? ¿Qué relación guarda con las bases de datos relacionales?

14.2. ¿Qué diferencias existen entre *MariaDB* y *MySQL*?

14.3. Descarga e instala el sistema gestor de base de datos *MariaDB*.

14.3. Creación de bases de datos

Una vez lanzado el asistente de instalación de *MariaDB* es fácil comprobar que, entre las aplicaciones disponibles para su instalación, se encuentra un cliente en línea de comandos. Este permitirá llevar a cabo todas las operaciones necesarias para crear bases de datos y sus correspondientes tablas. A continuación, se muestra un código SQL ejemplo lanzado sobre dicha herramienta para la construcción de una base de datos *Tienda* y la tabla *Producto* contenida en ella (figura 14.1).

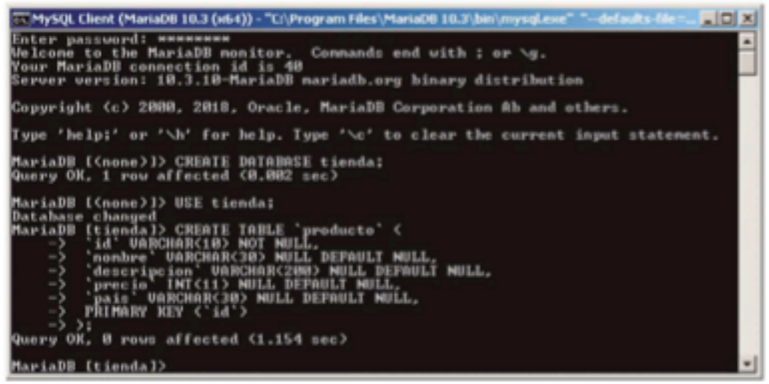


Figura 14.1
Creación de base de datos y tabla.

Recursos web www

En la URL https://www.w3schools.com/sql/ puedes encontrar un tutorial abierto sobre SQL.

Para conocer más sobre el manejo de *MariaDB*, puedes consultar la documentación en línea en https://mariadb.org/learn/. Si por el contrario has elegido *MySQL*, podrás consultarla en https://dev.mysql.com/doc/.

Actividad propuesta 14.4

Inicia el cliente cmd *MySQL/MariaDB* y construye la base de datos y la tabla del ejemplo.

14.4. Establecimiento de la conexión

Todo programa Java que tenga que interactuar con una base de datos necesitará de un *driver JDBC* con el que llevar a cabo la comunicación. Dicho *driver*, que es específico para cada SGBD, puede ser descargado desde la página oficial de cada uno de los fabricantes o proveedores de software y vendrá contenido en un fichero *jar*.

Recurso web

El *driver* para *MaríaDB* se puede obtener desde la URL https://downloads.mariadb.org/connector-java/.

Una vez descargado el *driver* deberá añadirse al proyecto como una librería.

Aunque se ha descargado el *driver* específico para *MariaDB*, *NetBeans* dispone ya entre sus librerías del *driver MySQL JDBC Driver*, que es perfectamente compatible.

Después de haber incluido el *driver* en el proyecto, se procederá a su registro desde el código del programa, que realizará la conexión con la base de datos mediante el método estático *Class.forName* indicando la ruta en la que puede ser contenido:

Figura 14.2
Inclusión de librería.

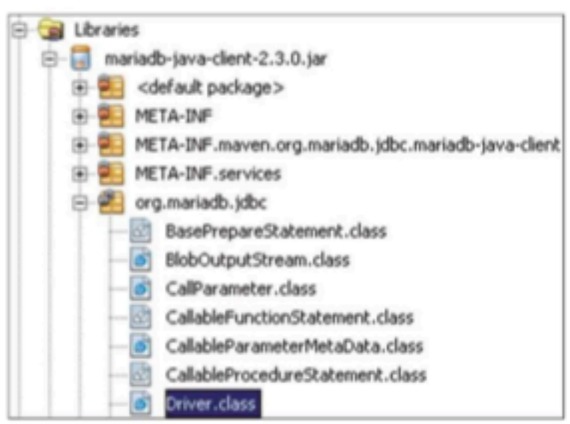


Figura 14.3
Ubicación del *driver* para emplear.


```
try {
    Class.forName("org.mariadb.jdbc.Driver");
} catch (java.lang.ClassNotFoundException e) {
    System.err.print("ClassNotFoundException: " + e.getMessage());
}
```

Habiéndose registrado el *driver* correctamente, el siguiente paso es construir la cadena de conexión. Aunque puede contener más parámetros que los indicados, basta con incluir el *subprotocolo* para el *motor* de base de datos con el que conectar (mariadb), el *host* (dirección IP, nombre DNS o nombre plano), el *puerto* de escucha (por defecto 3306) y la *base de datos* con la que conectar conforme a la sintaxis siguiente:

```
jdbc:mariadb://[host][:puerto]/[BD]
```

Esta cadena debe ser enviada como argumento al método *DriverManager.getConnection* junto con las credenciales del usuario con el que se realizará la conexión. Como resultado de dicha invocación se obtendrá el objeto *Connection*, que representará la conexión y a partir de él se podrá construir el objeto *Statement* que será el que realmente canalice las *query* hacia la base de datos. Una vez se hayan llevado a cabo todas las operaciones contra la base de datos será necesario cerrar la conexión. A continuación, se muestra un fragmento de código que realiza la conexión y desconexión contra una base de datos:

```
try {
   String url="jdbc:mariadb://localhost:3306/Tienda";
   Connection con = DriverManager.getConnection (url, user, pass);
   Statement stmt = con.createStatement();
   ... // operaciones CRUD
   stmt.close();
   con.close();
} catch(SQLException ex) {
   System.err.print("SQLException: " + ex.getMessage());
}
```

Actividad propuesta 14.5

Descarga el *driver* de *MariaDB* y construye un proyecto para conectar con la base de datos del ejemplo. A continuación, carga el *driver* descargado en el proyecto e implementa un programa que conecte con la base de datos y muestre el mensaje "Conexión exitosa" antes de cerrar la conexión.

14.5. Ejecución de consultas sobre la base de datos

En el capítulo anterior se ha hecho una pequeña introducción al lenguaje JPQL empleado en JPA cuya sintaxis es similar en muchos aspectos a SQL. Por este motivo no se entrará en detalle sobre las diferentes *query* lanzadas contra el sistema gestor de base de datos.

Para realizar cualquier tipo de operación se utilizará el objeto *Statement*. Este, a través del método *executeQuery*, ejecutará la consulta SQL que recibe como parámetro y recuperará los resultados en un objeto *ResultSet*.

Algunos de los métodos que ofrece *ResultSet* para el manejo de los resultados son:

- *getXXX ("campo" |columna)* → Obtiene la información de los campos de la fila actual conforme al tipo esperado de dato SQL. En la siguiente tabla se puede ver la correspondencia entre los tipos SQL y los métodos Java empleados para recuperarlos:

CUADRO 14.1
Métodos get de *ResultSet* para el manejo de resultados

getInt - INTEGER	getBigDecimal - DECIMAL	getDate - DATE
getLong - BIG INT	getBoolean - BIT	getTime - TIME
getFloat - REAL	getString - VARCHAR	getTimestamp - TIME STAMP
getDouble - FLOAT	getString - CHAR	getObject - OTROS

Estos métodos están sobrecargados y admiten como argumento tanto el nombre del campo (*String*) como el número de columna (la 1ª columna se corresponde con 1).

- *next()* → Para pasar al registro siguiente y seguir leyendo la información. Devuelve *false* cuando no hay más registros.
- *previous()* → Comportamiento inverso al de *next()*.
- *first()* → Para posicionarse en el primer registro.
- *last()* → Para posicionarse en el último registro.

Recurso web www

En la URL https://docs.oracle.com/javase/10/docs/api/java/sql/ResultSet.html podrás obtener información más completa sobre los muchos métodos disponibles para los objetos *ResultSet*.

A continuación, se muestra un ejemplo típico de consulta sobre la tabla producto de la base de datos tienda creada anteriormente. En él se lanza una *query* que devuelve todos los productos cuyo país es "España":


```
String query = "SELECT * FROM producto WHERE pais = 'Espana'";
ResultSet resultados = stmt.executeQuery(query);
System.out.println("Listado de productos Españoles: ");
while (resultados.next()) {
   System.out.println("Producto "+resultados.getString("nombre"));
}
```

14.6. Mecanismos de actualización de la base de datos

Por defecto todas las conexiones realizadas contra la base de datos mediante *DriverManager* son *autocommit*. Esto quiere decir que por cada transacción realizada (inserción, borrado o modificación), se hace el *commit* automáticamente y sin dar la posibilidad de cancelar la operación. Para evitarlo es posible ejecutar el método *setAutoCommit(false)* que obliga a ejecutar explícitamente el método *commit()* para hacer los cambios persistentes. Si por algún motivo se desea cancelar las operaciones, se puede hacer uso del método *rollback()*.

Mientras que para realizar consultas se emplea el método *executeQuery* del objeto *Statement*, para hacer transacciones se utiliza el método *executeUpdate*. Este método ejecutará la *query* SQL que recibe como parámetro y devolverá el número de filas afectadas. A continuación, se muestra un ejemplo sobre la tabla producto de la base de datos tienda en el que se realizan diferentes transacciones:

```
String query1 = "INSERT INTO `tienda`.`producto` (`id`, `nombre`, "
        + "`descripcion`, `precio`, `pais`) VALUES ('eee', 'frambuesa', "
        + "'frambuesa importación', '3', 'Portugal');";
int filasInsertadas = stmt.executeUpdate(query1);
```

```
String query2 = "UPDATE `tienda`.`producto` SET `precio`='6' WHERE "
        + "`id`='ddd';";
int filasAfectadas = stmt.executeUpdate(query2);
```

```
String query3 = "DELETE FROM `tienda`.`producto` WHERE `id`='eee';";
int filasEliminadas = stmt.executeUpdate(query3);
```

Realizar un número elevado de transacciones independientes puede afectar negativamente al rendimiento del sistema. Para esas situaciones es posible hacer un procesamiento por lotes agrupando las *querys* en el objeto *Statement* y enviándolas con una única llamada a la base de datos. Para ello habrá que seguir los pasos siguientes:

1. Establecer *auto-commit a false* mediante el método *setAutoCommit(boolean b)* del objeto *Connection*.
2. Agregar tantas *query* SQL al lote como sea necesario usando el método *addBatch (String sql)* del objeto *Statement* creado.
3. Ejecutar todas las sentencias de SQL usando el método *executeBatch()* del objeto *Statement*. El método devolverá un *array* en el que la primera posición contendrá el número de registros afectados por la primera *query*, la segunda posición el número de registros afectados por la segunda *query* y así sucesivamente.
4. Finalmente, confirmar todos los cambios usando el método *commit()* del objeto *Connection*.

Para vaciar todas las *query* añadidas se deberá hacer uso del método *clearBatch()* del objeto *Statement*.

Las *querys* del ejemplo anterior *(query1, query2 y query3)* podrían ejecutarse de la siguiente manera:

```
// Desactivando auto-commit
con.setAutoCommit(false);
// Añadiendo querys al lote
stmt.addBatch(query1);
stmt.addBatch(query2);
stmt.addBatch(query3);
// Recuperando el número de registros afectados
int[] numRegistrosAfectados = stmt.executeBatch();
// Aplicando cambios
con.commit();
```

Para saber más

Otro método empleado en realizar consultas o modificar los contenidos de la base de datos es *execute()*. Este método es válido para cualquier tipo de operación y devuelve un valor booleano. Si dicho valor es true indica que el método devolvió un objeto *ResultSet* que se puede recuperar utilizando el método *getResultSet;* si por el contrario el valor es *false* indica que la consulta devolvió un valor *int* o *void*

A continuación se muestran dos fragmentos de código equivalentes (uno con *executeQuery* y otro con *execute*):

```
stmt = conn.createStatement();
rs = stmt.executeQuery("SELECT * FROM producto");
```

```
if (stmt.execute("SELECT * FROM producto")) {
    rs = stmt.getResultSet();
}
```

Investiga

Investiga sobre las ventajas y el uso de PreparedStatement para lanzar querys contra la base de datos. Propón un ejemplo de uso.

Actividad propuesta 14.6

¿En qué circunstancias es útil deshabilitar *autocommit*? Justifica tu respuesta.

14.7. Recuperación de información

Además de los datos existentes en las diferentes tablas, los cuales son obtenidos mediante las operaciones típicas de consulta a través de sentencias del lenguaje de manipulación de datos (DML), también existe metainformación asociada que es importante conocer y saber recuperar. Por ejemplo, es posible saber qué usuario se está empleando para la conexión, qué bases de datos existen en el SGBD, el número máximo de conexiones admitidas…

Para poder recuperar toda esta metainformación se puede optar por una de estas alternativas:

1. Construir una *query* cuyo contenido sea una instrucción del SGBD que devuelva la metainformación en un *ResultSet* y procesarlo. Algunas de las funciones que se pueden emplear para construir la *query* son "SHOW TABLES", "SELECT USER()", "SELECT VERSION()", "DESCRIBE tabla"… Ejemplo:

```
String query = "SHOW DATABASES";
ResultSet resultados = stmt.executeQuery(query);
System.out.println("BD del sistema: ");
while (resultados.next()) {
    System.out.println("- " + resultados.getString(1));
}
```

```
run:
BD del sistema:
- information_schema
- mysql
- performance_schema
- test
- tienda
```

2. Recuperar la metainformación mediante el objeto *DatabaseMetaData* obtenido a partir del método *getMetaData* del objeto *Connection*. Este objeto dispone de métodos para la obtención de los metadatos. Algunos de ellos retornan un valor único mientras que otros devuelven un *ResultSet* que es necesario procesar. A continuación, se muestran algunos de estos métodos:

Cuadro 14.2
Métodos *get* de *DatabaseMetaData* para la recuperación de metainformación

`String getURL()`	Devuelve la URL de conexión.
`String getDriverName()`	Devuelve el nombre del *driver* empleado.
`String getDriverVersion()`	Devuelve la versión del *driver* empleado.
`String getDatabaseProductVersion()`	Devuelve la versión de SGBD empleado.
`ResultSet getCatalogs(…)`	Devuelve las BD del SGBD.
`ResultSet getTables(…)`	Devuelve las tablas de una BD.
`ResultSet getColumns(…)`	Devuelve las columnas de una tabla de la BD.

```
DatabaseMetaData metadata = con.getMetaData();
ResultSet resultados;
resultados = metadata.getCatalogs();
System.out.println("BD del sistema: ");
while (resultados.next()) {
    System.out.println("- " + resultados.getString("TABLE_CAT"));
}
```

```
run:
BD del sistema:
- information_schema
- mysql
- performance_schema
- test
- tienda
```

14.8. Manipulación de la información

No solamente se podrá recuperar información desde la base de datos, sino que también será posible modificarla mediante *querys* de la misma forma que si se realizase una operación INSERT, UPDATE o DELETE. Tal como puede ver en el ejemplo siguiente, los pasos a seguir para construir la *query* que manipule la información del sistema gestor de base de datos son similares a los seguidos en el apartado *14.6* (Mecanismos de actualización de la base de datos), pero reemplazando las sentencias del lenguaje de manipulación de datos (DML) por sentencias del lenguaje de definición de datos (DDL):

```
String query1 = "CREATE DATABASE empresa;";
String query2 = "USE empresa;" ;
String query3 = "CREATE TABLE IF NOT EXISTS `cliente` (\n"
        + "     `id` VARCHAR(10) NOT NULL,\n"
        + "     `nombre` VARCHAR(45) NOT NULL,\n"
        + "     `apellido1` VARCHAR(45) NOT NULL,\n"
        + "     `apellido2` VARCHAR(45) NOT NULL,\n"
        + "     PRIMARY KEY (`id`)\n"
        + ");";
stmt.executeUpdate(query1);
stmt.executeUpdate(query2);
stmt.executeUpdate(query3);
```

```
MySQL Client (MariaDB 10.3 (x64)) - "C:\Program Files\MariaDB 10.3\bin...
MariaDB [empresa]> DESCRIBE cliente;
+-----------+-----------+------+-----+---------+-------+
| Field     | Type      | Null | Key | Default | Extra |
+-----------+-----------+------+-----+---------+-------+
| id        | varchar(10) | NO   | PRI | NULL    |       |
| nombre    | varchar(45) | NO   |     | NULL    |       |
| apellido1 | varchar(45) | NO   |     | NULL    |       |
| apellido2 | varchar(45) | NO   |     | NULL    |       |
+-----------+-----------+------+-----+---------+-------+
4 rows in set (0.035 sec)

MariaDB [empresa]>
```

14.9. Utilización de asistentes

Además del SGBD, el instalador de *MariaDB* ofrece la posibilidad de instalar otras herramientas que pueden ser de gran utilidad. Una de ellas es el cliente en modo consola para *MySQL/MariaDB* utilizado anteriormente. Otra es *HeidiSQL*, un potente software gratuito que permite ver y editar datos y estructuras de los sistemas de bases de datos *MySQL/MariaDB*, *Microsoft SQL* o *PostgreSQL*.

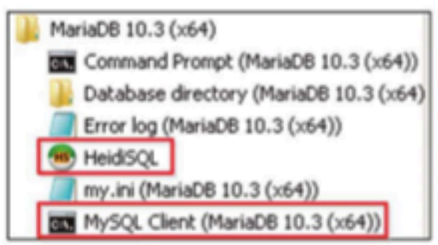


Figura 14.4
Asistentes y herramientas.

El empleo de *HeidiSQL* facilita incluso la generación de las *query*. Como puede verse en la figura 14.5, por cada operación realizada se muestra el código SQL equivalente en la ventana inferior. Además, ofrece la posibilidad de administrar diferentes sesiones y recordar los datos de conexión de estas, facilitando de esta manera las tareas de conexión y desconexión.

Otra de las herramientas disponibles para gestionar las bases de datos se encuentra en el *IDE NetBeans*. Se trata del gestor de servicios, desde el que se pueden gestionar los *drivers* empleados en las conexiones, así como las diferentes bases de datos con sus respectivas tablas, índices…

Por último, pero no por ello menos importante, también se puede hacer uso de la herramienta *MySQL Workbench*. Esta herramienta engloba las funcionalidades de las herramientas anteriores y añade algunas más. Se puede descargar libremente y dispone de gran cantidad de información sobre su uso en la URL https://dev.mysql.com/doc/workbench/en/. En la figura 14.7 puede verse la interfaz de la ventana principal de administración de la aplicación.

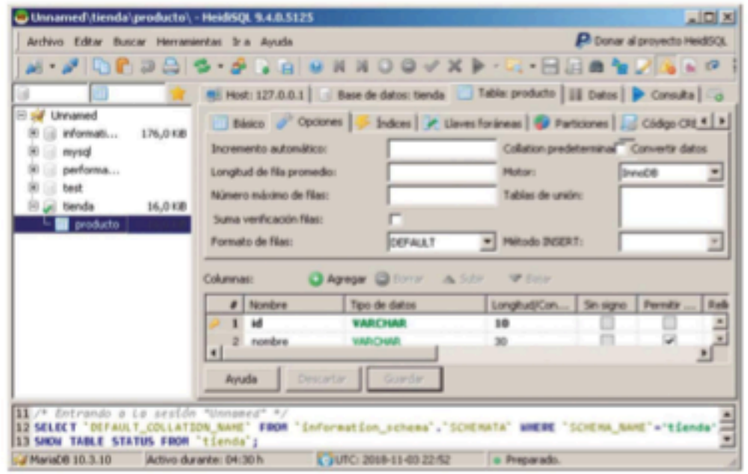


Figura 14.5
Manejo de *HeidiSQL*.

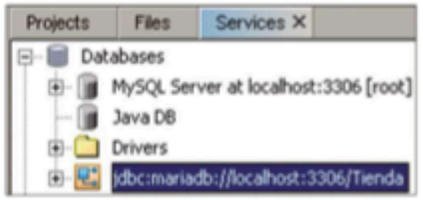


Figura 14.6
Gestor de servicios *NetBeans*.

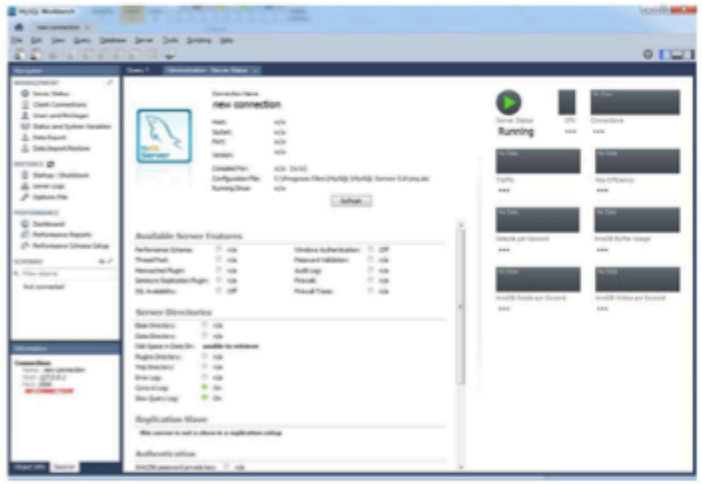


Figura 14.7
Ventana principal *MySQL Workbench*.

Resumen

- El DDL permite declarar las estructuras de almacenamiento de los datos sobre las que el DML realizará las operaciones CRUD.
- Los datos de una consulta se recogen en un *ResultSet* y desde ahí se pueden recuperar mediante los métodos *getXXX*, siendo *XXX* un tipo de dato.
- Además de la información almacenada en una base de datos, es posible recuperar la metainformación asociada a la misma y al SGBD.
- Existen varios asistentes que permiten trabajar de forma cómoda con las BD.
- La secuencia de pasos a seguir para conectar un programa Java con un SGBD es la siguiente:

  1. Descargar el *driver* que permite establecer la conexión desde Java al SGBD (JDBC).
  2. Añadir la librería con el *driver* al proyecto.
  3. Registrar el *driver* desde el código del programa.
  4. Construir la URL de conexión.
  5. Crear el objeto *Connection* con la URL de conexión.
  6. Crear el objeto *Statement* para lanzar las *query*.
  7. Ejecutar la *query* a través del *Statement* y recuperar los datos en un *ResultSet* (en caso de que sea una consulta).
  8. Liberar los recursos (cerrar el *Statement* y cerrar la conexión).

Ejercicios propuestos

1. ¿Qué diferencias y similitudes existen entre DML y DDL?
2. ¿Cuál de los métodos que permiten obtener metainformación desde Java indica el número máximo de conexiones que pueden ser atendidas simultáneamente?
3. ¿Qué método permite establecer el tiempo máximo de conexión?
4. Realiza una comparativa de las instrucciones *execute*, *executeQuery* y *executeUpdate*. Propón un ejemplo.
5. Realiza un estudio sobre los diferentes métodos *getXXX* de la clase *ResultSet* y propón ejemplos de uso.

Supuestos prácticos

1. Adapta el programa implementado en los supuestos prácticos 1 y 2 del capítulo 13 para que utilice una base de datos relacional sobre *MariaDB* o *MySQL*.
2. Adapta el programa implementado en los supuestos prácticos 3, 4 y 5 del capítulo 13 para que utilice una base de datos relacional sobre *MariaDB* o *MySQL*.
3. Implementa un programa que muestre los detalles de cada una de las tablas de las diferentes bases de datos del SGBD.
4. Implementa un programa Java que genere un informe en un fichero de texto con la metainformación relativa a la base de datos de los supuestos prácticos 1 y 2 (número de tablas, número de campos por tabla, tipos de los campos…).
5. Se desea disponer de un programa para gestionar la información de jugadores y equipos participantes en un campeonato de liga de fútbol. El programa permitirá dar de alta y de baja equipos y jugadores, así como consultar y modificar los datos de estos. La acción que se va a realizar se seleccionará desde un menú por consola con las opciones ya descritas más las de "Salir" y "Guardar". De cada equipo se almacenará su código de equipo (PK), nombre, estadio, ciudad y número de socios. De cada jugador se almacenará su código de jugador (PK), nombre, apellidos, fecha de nacimiento y equipo en el cual juega. Los datos que se vayan introduciendo serán almacenados en memoria principal y no se volcarán en la base de datos hasta que se seleccione la opción "Guardar" del menú de opciones.

   Implementa las clases necesarias para almacenar los datos anteriormente descritos y el código del programa principal.

Actividades de autoevaluación

1. ¿Cuál de las siguientes sentencias no pertenece a DML?
   - ☐ *a)* *UPDATE*.
   - ☐ *b)* *CREATE*.
   - ☐ *c)* Las opciones *a)* y *b)* son correctas.
   - ☐ *d)* Las opciones *a)* y *b)* son incorrectas.

2. De los métodos mostrados a continuación, ¿cuál permite cancelar las operaciones realizadas tras haber ejecutado *setAutoCommit(false)*?
   - ☐ *a)* *commit*().
   - ☐ *b)* *rollback*().
   - ☐ *c)* *forward*().
   - ☐ *d)* Todas las opciones anteriores son incorrectas.

3. ¿Cuál de los siguientes métodos no es empleado para lanzar una *query* contra la base de datos?
   - ☐ *a)* *executeQuery*.
   - ☐ *b)* *executeUpdate*.
   - ☐ *c)* *execute*.
   - ☐ *d)* Todas las opciones anteriores son incorrectas.

4. La clase empleada para lanzar *querys* contra la base de datos es:
   - ☐ *a)* *Class*.
   - ☐ *b)* *Connection*.
   - ☐ *c)* *Statement*.
   - ☐ *d)* Todas las opciones anteriores son incorrectas.

5. La clase empleada para registrar el *driver* con el que realizar la conexión a la BD es:
   - ☐ *a)* *Class*.
   - ☐ *b)* *Connection*.
   - ☐ *c)* *Statement*.
   - ☐ *d)* Todas las opciones anteriores son incorrectas.

6. ¿Cuál de las siguientes afirmaciones acerca de los métodos *getXXX* de *ResultSet* es verdadera?
   - ☐ *a)* Permiten acceder al contenido a través del nombre de la columna.
   - ☐ *b)* Permiten acceder al contenido a través del número de orden de la columna.
   - ☐ *c)* Las opciones *a)* y *b)* son correctas.
   - ☐ *d)* Todas las opciones anteriores son incorrectas.

7. Una tupla es:
   - ☐ *a)* Una fila o registro en una base de datos.
   - ☐ *b)* Una fila o registro en una tabla de la base de datos.
   - ☐ *c)* Una columna o atributo en una base de datos.
   - ☐ *d)* Una columna o atributo en una tabla de la base de datos.

8. ¿Cuál de las siguientes cadenas de conexión es válida para conectar con la base de datos BD localizada en el servidor *MariaDB* que corre en la máquina local y que escucha por el puerto 3306?
   - ☐ *a)* jdbc:mariadb:://localhost:3306/BD.
   - ☐ *b)* jdbc:mariadb:/localhost:3306/BD.
   - ☐ *c)* jdbc:mariadb://localhost:3306/BD.
   - ☐ *d)* jdbc:mariadb://localhost/3306/BD.

9. ¿Cuál es la clase del objeto a partir del que puede obtenerse el objeto *DatabaseMetaData*?
   - ☐ *a)* *Class*.
   - ☐ *b)* *Statement*.
   - ☐ *c)* *Connection*.
   - ☐ *d)* Todas las opciones anteriores son incorrectas.

10. ¿Cuál de los métodos siguientes permite obtener las bases de datos del SGBD?
    - ☐ *a)* *getCatalogs*.
    - ☐ *b)* *getTables*.
    - ☐ *c)* *getColumns*.
    - ☐ *d)* Todas las opciones anteriores son incorrectas.

SOLUCIONES:

1.	a	**b**	c	d	5.	**a**	b	c	d						
2.	a	**b**	c	d	6.	a	b	**c**	d	9.	a	b	**c**	d	
3.	a	b	c	**d**	7.	a	**b**	c	d	10.	**a**	b	c	d	
4.	a	b	**c**	d	8.	a	b	**c**	d						

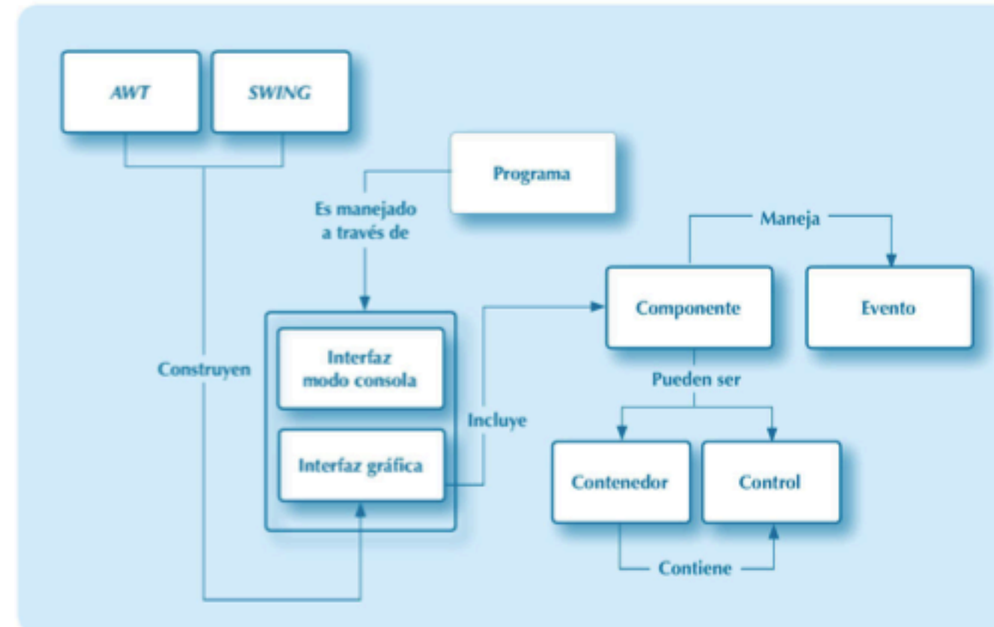
15

Creación de interfaces gráficas

Objetivos

- ✓ Conocer las tecnologías disponibles en Java para la creación de interfaces gráficas de usuario.
- ✓ Diseñar interfaces gráficas a partir de los diferentes elementos que pueden incluirse en una aplicación.
- ✓ Aprender a capturar los eventos producidos sobre un elemento y a enlazar dicha captura con el código encargado de manejarlos.
- ✓ Utilizar herramientas gráficas para el diseño de interfaces gráficas.

Mapa conceptual



Glosario

AWT. Siglas de *Abstract Window Toolkit* ("kit de herramientas de ventana abstracta", en español). Es un conjunto de herramientas diseñadas para la construcción de interfaces gráficas.

Embebido. Sinónimo de incrustado o contenido que se utiliza para representar al código o fragmento de texto que es ubicado dentro de otro código.

GUI. Siglas de *Graphical User Interface* ("interfaz gráfica de usuario", en castellano). Hace referencia a un programa, o parte de este, que permite al usuario interactuar con él u otros mediante el empleo del ratón y sobre un entorno visual más intuitivo que la consola.

Heavy weight. Traducido del inglés como "peso pesado", hace referencia a los componentes *AWT* que utilizan elementos nativos (del sistema operativo).

Homónimo. Se dice de algo que tiene el mismo nombre que otro u otra.

Intérprete de comandos. Programa que recibe texto a través de un terminal y lo traduce en instrucciones del sistema operativo.

JFC. Siglas de *Java Foundation Classes* ("clases base Java", en castellano). Es un conjunto elementos que permite crear interfaces gráficas de usuario con Java.

Layout. Esquema o distribución de los elementos contenidos en un diseño (entendiéndose como diseño un panel, una ventana, una pantalla, un plano…).

Light weight. Traducido del inglés como "peso ligero", hace referencia a los componentes utilizados por *Swing* para el desarrollo de aplicaciones con interfaz gráfica.

MS-DOS. Sistema operativo en modo consola de la familia Microsoft cuyas siglas provienen de *MicroSoft Disk Operating System*. Aunque su desarrollo y mantenimiento hoy en día está detenido, los sistemas operativos modernos Microsoft Windows incluyen un intérprete de comandos cuyo repertorio de instrucciones es similar al de dicho sistema.

Píxel. Término procedente del acrónimo inglés formado a partir de las palabras *picture* y *element*, que identifica a la unidad más pequeña en color que compone una imagen en formato digital.

Scrollbar. Barra de desplazamiento, horizontal o vertical, que es incluida en las ventanas de las aplicaciones con interfaz gráfica para acceder a las áreas de esta que hayan podido quedar ocultas por su disposición o tamaño.

15.1. Introducción

Hasta llegar a este capítulo, todos los ejemplos mostrados se han ejecutado sobre un terminal de Linux o bien sobre una consola del intérprete de comandos de MS-DOS. Aunque para determinados sistemas, cada vez menos, esta interfaz puede resultar suficiente, bien es cierto que la gran mayoría de las aplicaciones requieren de una interfaz gráfica (GUI) con la que el usuario pueda interactuar de una forma más cómoda y que permita el uso del ratón.

Como no podía ser de otra forma, Java ofrece el soporte necesario para la creación de aplicaciones con interfaz gráfica y lo hace proponiendo el uso de varias tecnologías. En los apartados siguientes se hará un breve recorrido por las que, históricamente, han sido las tecnologías más importantes para la creación de interfaces gráficas de usuario desde Java: *AWT* y *Swing*.

15.2. Librería *AWT*

Java *AWT* (*Abstract Window Toolkit*) es un API para el desarrollo de aplicaciones con interfaz gráfica desde Java cuyos componentes se encuentran en el paquete *java.awt*. Es la más primitiva de las tres que serán referenciadas en el capítulo.

AWT actualmente forma parte de JFC (*Java Foundation Classes*) y su característica principal es que utiliza clases comunes que posteriormente crean componentes nativos del sistema operativo, por lo que su aspecto siempre es dependiente del sistema operativo anfitrión sobre el cual se ejecute. Por este motivo son denominados *heavy weight*.

Para crear aplicaciones gráficas con *AWT* bastará con crear un proyecto e ir añadiendo los diferentes elementos gráficos sobre él.

15.2.1. Ventanas, contenedores y controles

Todos los elementos que se pueden ver en la interfaz gráfica se denominan *componentes* y heredan de la clase base *Component*. Cada componente implementa, al menos, los métodos recogidos en el cuadro 15.1.

Cuadro 15.1
Métodos de la clase base *Component*

`Dimension getSize();`	Devuelven el tamaño actual, máximo y mínimo del componente.
`Dimension getMaximumSize();`	
`Dimension getMinimumSize();`	
`void setSize(int ancho, int alto);`	Establecen el tamaño actual, máximo y mínimo del componente.
`void setMaximumSize(Dimension maximumSize);`	
`void setMinimumSize(Dimension minimumSize);`	
`Rectangle getBounds();`	Devuelve o establece las coordenadas y dimensiones del componente.
`void setBounds(int x, int y, int ancho, int largo);`	
`boolean getEnabled();`	Devuelve *true* si está activo.
`void setEnabled(boolean b);`	Activa o desactiva el componente.
`boolean getVisible();`	Retorna *true* si es visible o *false* en caso contrario.
`void setVisible(boolean b);`	Muestra u oculta el componente.

NOTAS: La clase *Dimension* tiene dos campos: *width* (anchura) y *heigth* (altura). La clase *Rectangle* tiene los campos: *x*, *y*, *width* y *heigth*.

Un *componente* puede ser un *contenedor* o bien un *control*. Entre los diferentes tipos de contenedor se encuentran:

1. *Container:* clase base del resto de contenedores (*Window*, *Frame*, *Dialog* o *Panel*).
2. *Window:* contenedor que no tiene bordes, título ni barras de menú. Es la clase base de *Frame* y *Dialog*, que son los que se utilizarán para crear una ventana. Este tipo de contenedor no puede estar contenido en ningún otro.
3. *Frame:* contenedor que tiene la barra de título y puede tener barras de menú.
4. *Dialog:* ventana con borde y título que puede incluirse dentro de otro contenedor. Puede ser *modal* en cuyo caso será la ventana activa de la aplicación y la única que pueda recoger eventos.

5. *FileDialog:* ventana de diálogo desde la cual el usuario puede seleccionar un archivo. Puede ser para cargar un fichero (*FileDialog.LOAD*) o bien para guardarlo (*FileDialog.SAVE*).
6. *Panel:* contenedor que no tiene ni la barra de título ni barras de menú, que debe ir contenido en un *Frame* y cuya única finalidad es agrupar controles dentro de él.

Algunos de los métodos de la clase *Container* se detallan en el cuadro 15.2.

Cuadro 15.2
Métodos de la clase base *Container*

`Component add(Component c);`	Añade un componente al contenedor.
`void print(Graphics g);`	Imprime el contenedor.
`void printComponents(Graphics g);`	Imprime cada uno de los componentes de este contenedor.
`void dispose()`	Libera todos los recursos nativos utilizados por esta ventana y los componentes contenidos. En el caso de que se trate de la última ventana de la aplicación, forzará el cierre de la JVM.
`int getComponentCount()`	Devuelve el número de componentes contenidos.
`Component getComponent(int n)`	Devuelve el enésimo componente contenido.
`Component getComponentAt(int x, int y)`	Devuelve el componente contenido en la posición indicada.
`Component[] getComponents()`	Devuelve un *array* de componentes contenidos.

Entre los controles que pueden ser incluidos en un contenedor están:

1. *Button:* es un botón con título. No tiene campos y su constructor permite pasar como argumento la cadena de título.
2. *Canvas:* elemento empleado para dibujar o capturar eventos del usuario.
3. *Checkbox:* botón de comprobación que puede o no estar seleccionado. Pueden agruparse para formar un radiobotón (*CheckboxGroup*), que es una agrupación de botones de comprobación de exclusión múltiple.
4. *Choice:* desplegable de opciones. El título del componente será el título del ítem que se encuentre seleccionado.
5. *Label:* etiqueta contenedora de texto.
6. *List:* listado desplegable visible en todo momento. Dispone de una *scrollbar* para visualizar los elementos que quedan ocultos.
7. *Scrollbar:* barra de desplazamiento que permite trabajar con rangos de valores.
8. *TextField:* componente que hereda de *TextComponent* empleado para introducir una línea de texto.
9. *TextArea:* componente que hereda de *TextComponent* empleado para introducir texto de más de una línea. Es posible configurar el componente para que muestre barras de desplazamiento vertical u horizontal (*SCROLLBARS_HORIZONTAL_ONLY*, *SCROLLBARS_VERTICAL_ONLY*, *SCROLLBARS_BOTH*, *SCROLLBARS_NONE*).

Un método que puede ser de gran utilidad es *getParent()*, que retorna la referencia del contenedor en el que se encuentra el control. El objeto devuelto será de la clase *Container*.

A continuación, se muestra un ejemplo básico de contenedor y controles:

```
package awt;
import java.awt.*;
public class AWT {
   public static void main(String[] args) {
        Frame marco = new Frame ("Frame AWT");
        marco.setLayout(null);
        marco.setSize(160, 160);
        marco.setVisible(true);
        // ----------------------------------------
        Button botonSalir = new Button("Salir");
        botonSalir.setBounds(30,30,50,30);
        marco.add(botonSalir);
        // ----------------------------------------------------------
        Button botonLimpiar = new Button("Limpiar");
        botonLimpiar.setBounds(80,30,50,30);
        marco.add(botonLimpiar);
        // ----------------------------------------------------------
        TextField cajaTexto = new TextField ("Escribe algo ...");
        cajaTexto.setBounds(30,70,100,30);
        marco.add(cajaTexto);
 }
}
```

Figura 15.1
Ejemplo de contenedor *Frame* y controles.

Actividad propuesta 15.1

¿Qué provoca utilizar valores negativos en un objeto de la clase *Dimension*? Justifica tu respuesta.

Investiga

Indaga acerca de para qué se utiliza el método *pack* de la clase *Frame* y propón un ejemplo de uso.

15.2.2. Eventos

Cuando un usuario efectúa una acción sobre un componente automáticamente se genera un objeto de una de las clases de eventos que lo identifica y que contiene información relativa al

mismo: tipo de evento, objeto en el que fue generado, momento en el que se ha producido, coordenadas en las que se produjo (ratón), teclas pulsadas (teclado)…

La jerarquía de clases de los objetos evento comienza en la clase *EventObject* que se encuentra contenida en el paquete *java.util*. La clase *AWTEvent*, que está contenida en el paquete *java.awt*, es una subclase de *EventObject* que representa la clase base de la que derivan todas las clases que representan los eventos que se pueden producir en aplicaciones basadas en *AWT*. Estas clases serán del tipo *XXXEvent*, en donde *XXX* hace referencia al tipo de evento: *ActionEvent*, *ComponentEvent*, *KeyEvent*, *MouseEvent*, *InputEvent*, *ItemEvent*, *TextEvent*, *WindowEvent*.

Si se ha llevado a cabo la codificación del ejemplo anterior, se habrá comprobado cómo cuando se intenta cerrar la ventana esta no responde y sigue abierta. Este comportamiento se debe a que no se ha añadido ningún capturador de los eventos al objeto y, por tanto, cualquier acción llevada a cabo sobre él (incluso la de cerrar la ventana) es ignorada. En el cuadro 15.3 se recoge la sintaxis general de los métodos empleados para añadir, eliminar o procesar eventos.

Cuadro 15.3
Sintaxis de los métodos para añadir eliminar o procesar eventos

`void add_XXX_Listener(_XXX_Listener l);`	Añade un capturador de eventos para algún tipo de eventos sobre este componente.
`void remove_XXX_Listener(_XXX_Listener l);`	Elimina algún capturador de eventos de este componente.
`void process_XXX_Event(_XXX_Event e);`	Procesa eventos del tipo *TipoEvent* enviándolos a cualquier objeto *TipoListener* que estuviera escuchando.

En los métodos anteriormente descritos *_XXX_* depende del tipo origen del evento que puede ser *Action*, *Component*, *Focus*, *InputMethod*, *Key*, *MouseMotion*…

A partir del ejemplo anterior en el que los botones incluidos no tenían ningún tipo de comportamiento asociado, se puede añadir el siguiente código para conseguir que el botón "Salir" fuerce el cierre de la aplicación, que el botón "Limpiar" vacíe el contenido del campo de texto y que el botón de cierre de la ventana fuerce también el cierre de la aplicación:

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
...
```



```
...
   marco.addWindowListener(new WindowAdapter(){
     @Override
     public void windowClosing(WindowEvent e) {
         marco.dispose();
       }
   });
   botonSalir.addActionListener(new ActionListener(){
     @Override
     public void actionPerformed(ActionEvent e) {
         ((Frame) botonSalir.getParent()).dispose();
       }
   });
   botonLimpiar.addActionListener(new ActionListener(){
     @Override
     public void actionPerformed(ActionEvent e) {
         ((TextField) marco.getComponentAt(30,70)).setText("");
       }
   });
...
```

Otro ejemplo puede ser el siguiente, en el que se abre un cuadro de diálogo para selección de ficheros tras pulsar en un botón:

```
Frame marco = new Frame();
Button b = new Button ("Abrir");
Label l = new Label ("No has seleccionado fichero.");
b.setBounds(30,30,50,30);
l.setBounds(90,30,220,30);
marco.add(b);
marco.add(l);

FileDialog fd = new FileDialog (marco, "Elige un fichero", FileDialog.LOAD);
fd.setDirectory("C:\\carpeta_ficheros");
fd.setFile("*.txt");
b.addActionListener (new ActionListener() {
 @Override
 public void actionPerformed(ActionEvent e) {
      fd.setVisible(true);
      if (fd.getFile()!=null) l.setText(fd.getDirectory()+fd.getFile());
   }
});
marco.setLayout(null);
marco.setSize (300,80);
marco.setVisible (true);
```

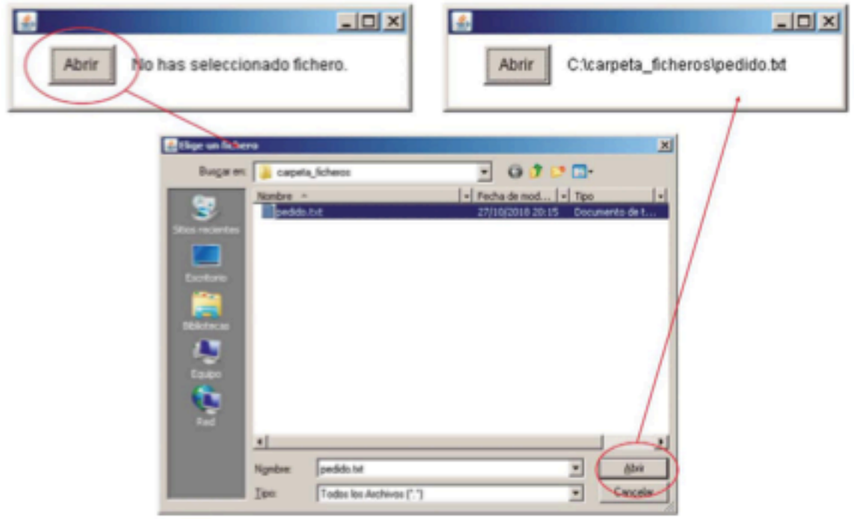


Figura 15.2
Ejemplo *FileDialog*.

15.2.3. *Layouts*

Todos los contenedores pueden tener asociado un *LayoutManager* que establece cómo serán distribuidos los componentes dentro de él. Si no se ha establecido, y por tanto su valor es *null*, el contenedor no usará ningún administrador de diseño, pudiéndose realizar de esta manera un posicionamiento absoluto. En este caso deberá especificar el tamaño y la posición de cada componente dentro del contenedor. La desventaja de utilizar posicionamiento absoluto es que, si cambia el tamaño del contenedor de nivel superior, los componentes contenidos no se ajustarán correctamente. Los métodos que permiten recuperar y establecer el *layout* deseado son mostrados en el cuadro 15.4.

Cuadro 15.4
Métodos para establecimiento y recuperación del *Layout*

`LayoutManager getLayout();`	Devuelve el gestor de impresión (*LayoutManager*) asociado a este contenedor, que es el responsable de colocar los componentes dentro del contenedor.
`void setLayout(LayoutManager l);`	Establece un gestor de impresión para este componente.

Existen varios tipos de *LayoutManager*:

1. *BorderLayout*: sitúa los componentes en cinco regiones cardinales: Norte, Sur, Este, Oeste y Centro. Cada región admite un único componente, pero nada impide que este sea, a su vez, otro contenedor que admita más componentes. Ejemplo:


```
Frame marco = new Frame ("BorderLayout");
marco.add (new Button("Norte"),"North");
marco.add (new Button("Sur"),"South");
marco.add (new Button("Este"),"East");
marco.add (new Button("Oeste"),"West");
marco.add (new Button("Centro"),"Center");
marco.setSize (200,100);
marco.setVisible(true);
```

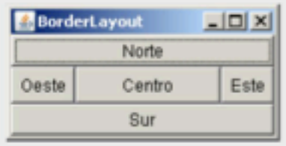


Figura 15.3
Disposición *BorderLayout*.

2. *CardLayout*: gestiona varios componentes que comparten el mismo espacio de visualización. La decisión de mostrar uno u otro se puede tomar según la acción realizada sobre otro control. Ejemplo:

```
Frame marco = new Frame ("CardLayout");
CardLayout cardLayout = new CardLayout();
marco.setLayout(cardLayout);
marco.setSize (200,80);
marco.setVisible (true);
Button boton1 = new Button ("Botón en página 1");
Button boton2 = new Button ("Botón en página 2");
marco.add (boton1);
marco.add (boton2);
boton1.addActionListener(new ActionListener() {
 @Override
 public void actionPerformed(ActionEvent e) {
     cardLayout.next(marco);
   }
});
boton2.addActionListener(new ActionListener(){
 @Override
 public void actionPerformed(ActionEvent e) {
     cardLayout.next(marco);
   }
});
```

Figura 15.4
Disposición *CardLayout*.

3. *FlowLayout*: dispone los componentes horizontalmente de izquierda a derecha en cada línea, pudiéndose establecer el espacio horizontal y vertical entre cada uno. Si se sobrepasa una línea comienza a la izquierda de la siguiente. Puede instanciarse con posicionamiento centrado (*FlowLayout.CENTER*), izquierdo (*FlowLayout.LEFT*), derecho (*FlowLayout.RIGHT*)... Ejemplo:

```
Frame marco = new Frame ("FlowLayout");
FlowLayout flowLayout = new FlowLayout(FlowLayout.CENTER,2,2);
marco.setLayout(flowLayout);
marco.setSize (200,80);
marco.setVisible (true);
Button boton1 = new Button ("boton1");
Button boton2 = new Button ("boton2");
Button boton3 = new Button ("boton3");
marco.add (boton1);
marco.add (boton2);
marco.add (boton3);
```

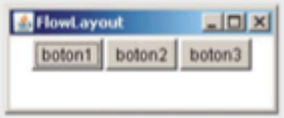


Figura 15.5
Disposición *FlowLayout*.

4. *GridLayout*: sitúa los componentes en una rejilla cuyas casillas tienen el mismo tamaño. Ejemplo:

```
Frame marco = new Frame ("GridLayout");
GridLayout gridLayout = new GridLayout(4,2);
marco.setLayout(gridLayout);
marco.setSize (200,80);
marco.setVisible (true);
Button b11 = new Button ("11"); Button b12 = new Button ("12");
Button b13 = new Button ("13"); Button b14 = new Button ("14");
Button b21 = new Button ("21"); Button b22 = new Button ("22");
Button b23 = new Button ("23"); Button b24 = new Button ("24");
marco.add (b11); marco.add (b12);marco.add (b13); marco.add (b14);
marco.add (b21); marco.add (b22);marco.add (b23); marco.add (b24);
```

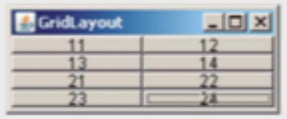


Figura 15.6
Disposición *GridLayout*.

Actividades propuestas

15.2. ¿Qué disposición se establece con *FlowLayout.LEADING* y *FlowLayout.TRAILING*?

15.3. Además de los *layout* mostrados, existe otro que utiliza una rejilla cuyas casillas pueden tener tamaño variable, ¿cuál es?

15.2.4. Menús

Los menús son los componentes que permiten organizar las diferentes acciones a llevar a cabo en una aplicación, pero su diseño depende en gran medida del tipo de aplicación a implementar, así como de las opciones que se quieran incluir.

Existen varias clases para el manejo de menús y todas ellas derivan de *MenuComponent*:

- ✓ *MenuBar*: barra de menú en la que se añaden objetos de tipo *Menu*.
- ✓ *Menu*: es el componente menú en el que se incluyen los diferentes *MenuItem*.
- ✓ *MenuItem*: representa una opción del menú.
- ✓ *MenuShortcut*: atajo de teclado asociado a un *MenuItem*. Hace uso de eventos *KeyEvent* para controlar si se pulsan determinadas teclas.
- ✓ *PopupMenu*: menú emergente asociado a un componente.
- ✓ *CheckboxMenuItem*: opción del menú que puede quedarse marcada al seleccionarse.

A continuación, se muestra un ejemplo de uso de los componentes descritos:


```
//Frame y barra de menú
marco = new Frame("Ejemplo Menú");
menuBar= new MenuBar();

//Menú "Archivo"
menu1 = new Menu("Archivo");
item1 = new MenuItem("Nuevo", new MenuShortcut(KeyEvent.VK_N,true));
item2 = new MenuItem("Abrir");
item3 = new MenuItem("Guardar", new MenuShortcut(KeyEvent.VK_G,true));
menu1.add(item1);
menu1.add(item?);
menu1.add(item?);
menu2 = new Menu("Guardar como");
item4 = new MenuItem(".pdf");
item5 = new MenuItem(".xlsx");
menu2.add(item4);
menu2.add(item5);
menu1.add(menu2);

//Menú "Otras"
menu3 = new Menu("Otras");
item6 = new MenuItem("Salir");
menu3.add(item6);

//Añadir "Archivo" y "Otras" a barra de menús
menuBar.add(menu1);
menuBar.add(menu3);

//Añadir barra de menús al marco
marco.setMenuBar(menuBar);

marco.setSize(220,200);
marco.setVisible(true);
```

Figura 15.7
Ejemplo menú.

Investiga

Busca información sobre la diferencia existente entre cerrar una aplicación mediante *System.exit(0)* o invocando al método *dispose* de la ventana principal.

Actividades propuestas

15.4. Obtén un listado de los diferentes *KeyEvent* disponibles para saber cómo manejarlos.

15.5. Busca información sobre el componente *PopupMenu* y realiza un programa que únicamente pueda cerrarse mediante una opción situada en este menú.

15.3. Librería *Swing*

Dadas las limitaciones que *AWT* tenía para representar determinados elementos (*barras de progreso, elementos árbol, tablas*…) se decidió evolucionar esta tecnología y fue ahí donde apareció *Java Swing*.

Mientras *AWT* hace uso de los controles nativos del sistema operativo anfitrión y es este el que captura los eventos, *Swing* "dibuja" los diferentes controles y hace que sea el propio Java quien los maneje. De esta manera los controles *Swing* para el sistema operativo serán simplemente píxeles, lo cual garantiza que las aplicaciones se ven exactamente igual independientemente del sistema operativo, así como que no haya comportamientos que puedan diferir de uno a otro.

Sabías que…

El lema "Escribe una vez, ejecuta en todas partes" fue duramente criticado cuando de aplicaciones *AWT* se trataba, pues este tipo de aplicaciones en ocasiones requerían de ciertas personalizaciones para los sistemas operativos sobre los que iban a trabajar.

15.3.1. Ventanas, contenedores y controles

Muchos de los controles anteriormente descritos tienen su correspondencia dentro del paquete *javax.swing*. Este tipo de controles, denominados *Light weight*, generalmente vienen identificados por un nombre precedido de la letra *J* y todos ellos, al igual que en *AWT*, heredan de la clase base *Component*.

En *Swing* también se puede hacer una primera distinción entre componentes contenedor o control. Los principales contenedores son *JFrame, JDialog* y *JPanel*. Se trata de clases derivadas de las empleadas en *AWT*, excepto *JPanel* que extiende de *JComponent*. A continuación se muestra un ejemplo básico de uso de contenedor y controles:

```
package swing;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JTextField;
public class SWING {
   public static void main(String[] args) {
      JFrame marco = new JFrame("JFrame Swing");
      JButton boton = new JButton("Haga clic");
      boton.setBounds(10,10,100,25);
      JTextField cuadroTexto = new JTextField();
      cuadroTexto.setBounds(120,10,100,25);
      boton.addActionListener(new ActionListener(){
          public void actionPerformed(ActionEvent e){
              cuadroTexto.setText("Hola mundo !");
          }
      });
```

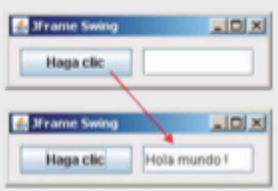


Figura 15.8
Ejemplo de contenedor *JFrame* y controles.


```
      marco.getContentPane().add(boton);
      marco.getContentPane().add(cuadroTexto);
      marco.setSize(240,70);
      marco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      marco.setLayout(null);
      marco.setVisible(true);
   }
}
```

Si se ha replicado y probado el código del ejemplo, se habrá comprobado cómo a diferencia de *AWT* ahora se permite el cierre de la aplicación únicamente configurando el comportamiento del cierre mediante método *setDefaultCloseOperation* y sin necesidad de añadir de forma explícita ningún manejador de eventos.

En cuanto a los controles disponibles en *Swing*, es importante destacar que existe un amplio repertorio formado por aquellos equivalentes a los ya existentes en *AWT* al que se añaden otros nuevos. Algunos de esos controles añadidos son:

1. *JColorChooser*: permite crear un cuadro de diálogo para que el usuario pueda seleccionar un color.

```
JFrame marco = new JFrame("Ejemplo JColorChooser");
JButton boton = new JButton("Seleccione color");
boton.setBounds(10,10,130,25);
boton.addActionListener(new ActionListener(){
   public void actionPerformed(ActionEvent e){
      Color color = Color.MAGENTA;
      color = JColorChooser.showDialog(marco,"Seleccione color",color);
      marco.getContentPane().setBackground(color);
   }
});
marco.getContentPane().add(boton);
marco.setSize(240,70);
marco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
marco.setLayout(null);
marco.setVisible(true);
```

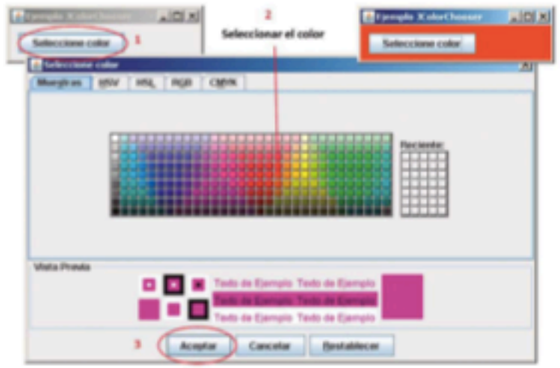


Figura 15.9
Ejemplo *JColorChooser*.

2. *JComboBox*: control desplegable que muestra un listado de opciones seleccionables.
3. *JList<E>*: muestra un listado de objetos permitiendo al usuario seleccionar uno o más ítems. Un ejemplo se puede ver en el uso de *BoxLayout*.
4. *JPasswordField*: proporciona un campo de texto para la entrada de contraseñas que sustituye cada carácter que escribe el usuario por un asterisco "*".
5. *JProgressBar*: representa la barra de progreso empleada para mostrar gráficamente el avance de un determinado proceso.
6. *JSeparator*: permite incluir líneas de división verticales u horizontales a menús y paneles.
7. *JSpinner*: cuadro de texto que permite al usuario escribir un valor y que contiene dos botones pequeños que posibilitan incrementar o decrementar el valor contenido.
8. *JSplitPane*: muestra un panel dividido en dos (uno al lado del otro o uno encima del otro). Es posible anidar varios *JSplitPane* para conseguir nuevos diseños.
9. *JTabbedPane*: permite al usuario seleccionar entre un grupo de componentes haciendo clic en una solapa.

```
JFrame marco = new JFrame("Ejemplo JTabbedPane");
JTabbedPane panelSolapas = new JTabbedPane();
// ------------------------- Creando panel JTable -------------------------
String informacion[][] = {
   {"11111111X","PACO"}, {"22222222Y","ROSA"}, {"33333333Z","LUISA"}
};
String columnas[] = {"DNI","NOMBRE"};
JTable tabla = new JTable(informacion,columnas);
JScrollPane panelJTable = new JScrollPane(tabla);
panelSolapas.add("JTable",panelJTable);
// ------------------------- Creando panel JTree -------------------------
JPanel panelJTree = new JPanel();
DefaultMutableTreeNode raiz = new DefaultMutableTreeNode("árbol");
DefaultMutableTreeNode nodo1 = new DefaultMutableTreeNode("1");
DefaultMutableTreeNode nodo1_1 = new DefaultMutableTreeNode("1_1");
DefaultMutableTreeNode nodo2 = new DefaultMutableTreeNode("2");
raiz.add(nodo1);raiz.add(nodo2);
nodo1.add(nodo1_1);
JTree arbol = new JTree(raiz);
panelJTree.add(arbol);
panelSolapas.add("JTree",panelJTree);
// Añadiendo panel de solapas con los paneles al marco
marco.getContentPane().add(panelSolapas);
marco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
marco.setSize(210,140);
marco.setVisible(true);
```

Figura 15.10
Ejemplo *JTabbedPane*, *JPanel*, *JTable* y *JTree*.

10. *JTable*: representa un control tabla en el que los datos se muestran por filas y columnas.
11. *JToolBar*: barra de herramientas que permite agrupar otros controles (normalmente botones con iconos).
12. *JTree*: permite representar jerarquías de datos en forma de árbol pero sin contenerlos en realidad (es una mera representación).
13. Otros controles *Swing* son: *JRadioButton*, *JScrollPane*, *JSlider* y *JToggleButton*.

15.3.2. Eventos

Los componentes *Swing* funcionan igual que los componentes *AWT* en cuanto a eventos se refiere, es por ello por lo que no conviene perder de vista *AWT*. Si se ha llegado a este apartado sin haber consultado anteriormente el dedicado a eventos en *AWT* (*15.2.2. Eventos*), sería muy conveniente retroceder a él para obtener una explicación más detallada sobre su manejo.

www **Recursos web**

En la URL https://docs.oracle.com/javase/tutorial/uiswing/examples/components/index.html puedes obtener ejemplos de los diferentes componentes *Swing*.

En la URL https://docs.oracle.com/javase/tutorial/uiswing/events/api.html podrás obtener información detallada sobre los diferentes tipos de evento disponibles en *Swing*.

Actividad propuesta 15.6

¿Qué eventos relativos al uso del teclado pueden capturarse sobre un botón? Prepara una tabla resumen.

15.3.3. Layouts

La disposición de los controles en una interfaz gráfica puede ser una de las tareas que más complicaciones presente al programador. En *Swing* existen varios tipos de *LayoutManager*, siendo algunos similares a los ya mostrados en *AWT* mientras que otros son nuevos o presentan alguna novedad con respecto a sus predecesores:

1. *BorderLayout*: es el *layout* predefinido. Al igual que su homónimo en *AWT*, también divide el área en 5 regiones, que en este caso son denominadas: PAGE_START, PAGE_END, LINE_START, LINE_END y CENTER. Ejemplo:

```
JFrame marco = new JFrame("BorderLayout");
marco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
JLabel etiqueta_PAGE_START = new JLabel ("PAGE_START");
marco.getContentPane().add(etiqueta_PAGE_START, BorderLayout.PAGE_START);
JLabel etiqueta_LINE_START = new JLabel ("LINE_START");
marco.getContentPane().add(etiqueta_LINE_START, BorderLayout.LINE_START);
JLabel etiqueta_CENTER = new JLabel ("CENTER");
etiqueta_CENTER.setPreferredSize(new Dimension(100, 50));
marco.getContentPane().add(etiqueta_CENTER, BorderLayout.CENTER);
JLabel etiqueta_LINE_END = new JLabel ("LINE_END");
marco.getContentPane().add(etiqueta_LINE_END, BorderLayout.LINE_END);
JLabel etiqueta_PAGE_END = new JLabel ("PAGE_END");
marco.getContentPane().add(etiqueta_PAGE_END, BorderLayout.PAGE_END);

for (Component etiqueta : marco.getContentPane().getComponents())
     ((JLabel)etiqueta).setHorizontalAlignment(JLabel.CENTER);

marco.pack();
marco.setVisible(true);
```

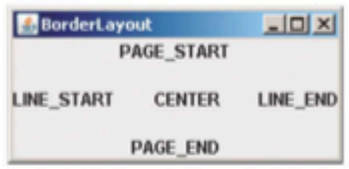


Figura 15.11
Disposición *BorderLayout*.

2. *BoxLayout*: permite que varios componentes se distribuyan vertical u horizontalmente en el mismo orden en que son añadidos. Se construye con un parámetro eje que especifica el tipo de diseño que se realizará. Hay cuatro opciones:

   a) *X_AXIS*: dispone los componentes horizontalmente de izquierda a derecha.
   b) *Y_AXIS*: sitúa los componentes verticalmente de arriba abajo.
   c) *LINE_AXIS*: los componentes se presentan en línea y orientados según la propiedad *ComponentOrientation* del contenedor.
   d) *PAGE_AXIS*: los componentes se distribuyen de la misma manera que las líneas de texto en una página, según la propiedad *ComponentOrientation* del contenedor.

```
JFrame marco = new JFrame("BoxLayout");
marco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

//Panel etiqueta y listado
JPanel panelListado = new JPanel();
panelListado.setLayout(new BoxLayout(panelListado, BoxLayout.PAGE_AXIS));
panelListado.setComponentOrientation(ComponentOrientation.RIGHT_TO_LEFT);
panelListado.setBorder(BorderFactory.createTitledBorder("Panel listado"));
// -- Creando componentes etiqueta y listado
```



```
JLabel etiqueta1 = new JLabel("listado");
JLabel etiqueta2 = new JLabel("números");
String[] data = {"Uno", "Dos", "Tres", "Cuatro", "Cinco"};
JList<String> lista = new JList<>(data);
JScrollPane listScroller = new JScrollPane(lista);
listScroller.setPreferredSize(new Dimension(240, 70));
listScroller.setAlignmentX(LEFT_ALIGNMENT);
// -- Añadiendo componentes con un área fijo de 5px de separación
panelListado.add(etiqueta1);
panelListado.add(etiqueta2);
panelListado.add(Box.createRigidArea(new Dimension(0,5)));
panelListado.add(listScroller);

//Panel botonera con botones a ambos lados.
JPanel botonera = new JPanel();
botonera.setLayout(new BoxLayout(botonera, BoxLayout.LINE_AXIS));
botonera.setBorder(BorderFactory.createTitledBorder("Panel botonera"));
// -- Creando componentes botón
JButton boton1 = new JButton ("boton1");
JButton boton2 = new JButton ("boton2");
// -- Añadiendo componentes con un área de separación
botonera.add(boton1);
botonera.add(Box.createHorizontalGlue());
botonera.add(boton2);

//Añadir los dos paneles al marco y hacerlo visible
marco.getContentPane().add(panelListado, BorderLayout.PAGE_START);
marco.getContentPane().add(botonera, BorderLayout.PAGE_END);
marco.pack();
marco.setVisible(true);
```

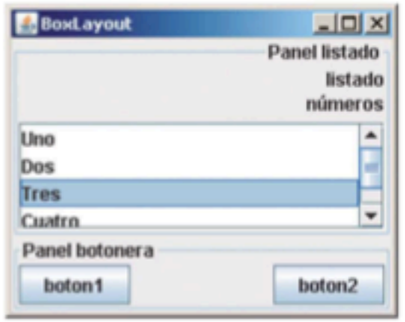


Figura 15.12
Disposición *BoxLayout*.

Sabías que...

La clase Box permite añadir espacios entre los controles contenidos que pueden ser fijos (*Box.createRigidArea*) o que tomen todo el espacio sobrante (*Box.createHorizontalGlue*).

3. *GroupLayout*: fue desarrollado para ser utilizado por herramientas de construcción de *GUI*, pero también puede usarse manualmente. Permite indicar cómo disponer los controles mediante diseños horizontales y verticales totalmente independientes. Como requisito, todos los componentes deben ser definidos dos veces (una por cada dimensión); de no hacerse provocará una excepción.

   Dentro de cada dimensión los controles pueden disponerse de forma:

   a) *Secuencial*: se colocan uno tras de otro (de forma similar a *BoxLayout* o *FlowLayout*). La posición de cada componente será relativa al componente anterior.
   b) *Paralela*: se sitúan uno encima del otro en el mismo espacio. Pueden alinearse mediante:

      - CENTER: los elementos deben estar alineados al centro de la región.
      - BASELINE: los elementos deben estar alineados a lo largo de su línea de base.
      - LEADING: los elementos deben alinearse al origen de la región. Para el eje horizontal con orientación de izquierda a derecha, significa que está alineado con el borde izquierdo. Para el eje vertical está alineado con el borde superior.
      - TRAILING: los elementos deben estar alineados al final de la región. Para el eje horizontal con orientación de izquierda a derecha, significa que está alineado con el borde derecho. Para el eje vertical está alineado con el borde inferior.

   Las alineaciones en el eje horizontal únicamente serán perceptibles si los componentes no son todos del mismo tamaño.

   Los controles podrán agruparse (*addGroup*) y disponerse en el grupo conforme a una de las disposiciones descritas anteriormente (secuencial o paralela), pero un agrupamiento o disposición erróneos provocará que queden superpuestos entre sí. A continuación, se muestra un ejemplo de uso de este *layout* con distintos agrupamientos y alineaciones:

```
// creación de componentes
JLabel label1 = new JLabel("etiqueta1");
JLabel label2 = new JLabel("etiqueta2");
JLabel label3 = new JLabel("etiqueta3_");
JLabel label4 = new JLabel("_etiqueta4");
JLabel label5 = new JLabel("etiqueta5");
JLabel label6 = new JLabel("etiqueta6");
JLabel label7 = new JLabel("etiqueta7");
JLabel label8 = new JLabel("etiqueta8");

JFrame marco = new JFrame ("GroupLayout");
GroupLayout layout = new GroupLayout(marco.getContentPane());
marco.getContentPane().setLayout(layout);
layout.setAutoCreateGaps(true);
layout.setAutoCreateContainerGaps(true);

// configurando diseño horizontal
layout.setHorizontalGroup(layout.createSequentialGroup()
    .addComponent(label1)
    .addGroup(layout.createParallelGroup(LEADING)
      .addComponent(label7)
      .addComponent(label8)
    )
    .addGroup(layout.createParallelGroup(CENTER)
      .addComponent(label2)
      .addGroup(layout.createSequentialGroup()
          .addGroup(layout.createParallelGroup(TRAILING)
              .addComponent(label3)
              .addComponent(label5)
          )
          .addGroup(layout.createParallelGroup(LEADING)
              .addComponent(label4)
              .addComponent(label6)
          )
      )
    )
);

// configurando diseño vertical
layout.setVerticalGroup(layout.createSequentialGroup()
    .addGroup(layout.createParallelGroup(BASELINE)
      .addComponent(label1)
      .addComponent(label2)
    )
    .addGroup(layout.createParallelGroup(LEADING)
      .addComponent(label8)
      .addGroup(layout.createSequentialGroup()
          addGroup(layout.createParallelGroup(BASELINE)
              .addComponent(label3)
              .addComponent(label4))
          .addGroup(layout.createParallelGroup(BASELINE)
              .addComponent(label5)
              .addComponent(label6)
          )
      )
    )
    .addComponent(label7)
);

marco.pack();
marco.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
marco.setVisible(true);
```

Figura 15.13
Esquema disposición *GroupLayout*.

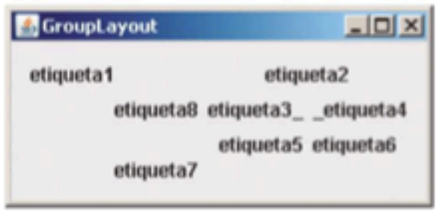


Figura 15.14
Disposición *BoxLayout*.

4. *SpringLayout*: permite especificar relaciones de distancia horizontal y vertical entre los bordes de los componentes para realizar un posicionamiento relativo. Para ello es necesario hacer uso del método *putConstraint* al cual se le pasa como argumento las referencias de los objetos dependiente y ancla, los bordes de esos objetos a tener en cuenta para el posicionamiento y la distancia entre ambos. A continuación, se muestra un ejemplo de uso:

```
JFrame marco = new JFrame("SpringLayout");
Component b1 = new JButton("botón 1");
Component b2 = new JButton("botón 2");
Component b3 = new JButton("botón 3");

marco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
marco.setSize(245,90);

Container panel = marco.getContentPane();
SpringLayout layout = new SpringLayout();
marco.setLayout(layout);

marco.add(b1);
marco.add(b2);
marco.add(b3);

layout.putConstraint(SpringLayout.WEST, b1, 0, SpringLayout.WEST, pa-
nel);
layout.putConstraint(SpringLayout.NORTH, b1, 0, SpringLayout.NORTH, pa-
nel);
layout.putConstraint(SpringLayout.WEST, b2, 0, SpringLayout.EAST, b1);
layout.putConstraint(SpringLayout.NORTH, b2, 0, SpringLayout.SOUTH, b1);
layout.putConstraint(SpringLayout.EAST, b3, 0, SpringLayout.EAST, pa-
nel);
layout.putConstraint(SpringLayout.NORTH, b3, 5, SpringLayout.SOUTH, b1);

marco.setVisible(true);
```

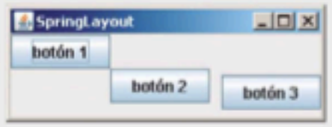


Figura 15.15
Disposición *SpringLayout*.

5. Otros *layouts* disponibles y similares a los de *AWT* son *CardLayout*, *FlowLayout*, *GridBagLayout* y *GridLayout*.

INVESTIGA

Busca información sobre la funcionalidad de los métodos *setAutoCreateGaps* y *setAutoCreateContainerGaps*.

15.3.4. Modificando la apariencia

Puesto que en *Swing* los controles son propios de Java, excepto la ventana sobre la que se ejecuta la aplicación, que es del sistema operativo, es muy fácil cambiar la apariencia con la que serán mostrados mediante una característica denominada *Look & Feel*. Para conocer los diferentes estilos disponibles en el JDK instalado, se puede ejecutar el siguiente código:

```
UIManager.LookAndFeelInfo[] looks = UIManager.getInstalledLookAndFeels();
for (UIManager.LookAndFeelInfo look : looks) {
    System.out.println(look.getClassName());
}
```

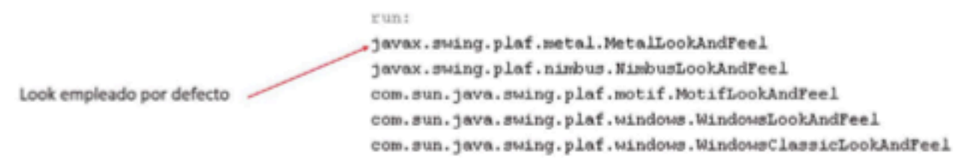


Figura 15.16
Estilos disponibles en el JDK instalado.

Y para hacer uso de cada uno de ellos será necesario cargar el estilo mediante el método *setLookAndFeel* del gestor de interfaces de usuario (*UIManager*) añadiendo la línea siguiente al comienzo del programa (en donde *look* es un *String* que identifica el estilo a aplicar):

```
UIManager.setLookAndFeel(look);
```

Además, las ventanas que se vayan creando podrán ser decoradas si el estilo seleccionado soporta la decoración de ventanas y si se ha indicado que se desea realizar. Para ello es preciso añadir al código las instrucciones:

```
JFrame.setDefaultLookAndFeelDecorated(true);
JDialog.setDefaultLookAndFeelDecorated(true);
```

Por otro lado, también es posible customizar algunos controles. Por ejemplo, la cadena de texto que se muestra en algunos controles (botón, label…) es escrita de forma predeterminada con una sola fuente, un color y en una única línea. Mediante los métodos *setFont* y *setForeground* del componente se podrá modificar la fuente y su color, respectivamente. Pero si además se desea mezclar fuentes o colores en el texto, o si desea escribirlo en varias líneas, es posible usar como cadena un código HTML *embebido*.

A continuación se muestran ejemplos de aplicaciones de estilos (figura 15.17) así como de una decoración mediante HTML embebido (figura 15.18).

Figura 15.17
Aplicación de estilos y decoraciones disponibles.

```
JButton boton;
boton = new JButton("<html>"
        + "<b>Haga</b>"
        + "<br>"
        + "<font color=red>clic</font>"
        + "</html>");
```

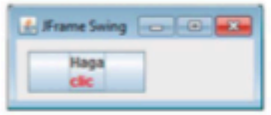


Figura 15.18
Decoración con HTML embebido.

Para saber más

Es posible mezclar componentes *AWK* y *Swing* en la misma aplicación sin grandes problemas desde la versión *JDK 6 Update 12*.

JavaFX es la tecnología empleada en el desarrollo de aplicaciones gráficas de cliente enriquecido que pueden correr en varias plataformas (*smartphones*, televisores, ordenadores de escritorio…). A diferencia de las anteriores (*AWT* y *Swing*), todas las funciones necesarias para el desarrollo residen en una sola biblioteca. Gracias a la clase *Swing Node* se puede incrustar contenido *Swing*.

Investiga

Indaga sobre cómo establecer la apariencia con la que ejecutar una aplicación desde línea de comandos.

15.4. Herramientas gráficas

Como se habrá podido comprobar, la creación y configuración de interfaces gráficas desde el código fuente es una tarea tediosa. Tanto si se han construido los contenedores y controles di-

rectamente en el programa principal, como si se ha extendido su clase principal de alguna de las clases contenedor y se ha instanciado en su método *main*, la disposición y configuración de sus componentes habrá tenido que realizarse desde el propio código fuente. Gracias a los asistentes gráficos, esta tarea se puede realizar de forma visual y en muchos menos pasos.

NetBeans ofrece todo lo necesario para construir aplicaciones *GUI* tanto con componentes *AWT* como con componentes *Swing*. Los pasos siguientes son una guía para todo el proceso:

1. Construye un nuevo proyecto vacío (sin clase principal) para la aplicación.
2. Sobre el proyecto añade un *frame* para la aplicación. Podrás elegir entre un *frame AWT* o uno *Swing*. En cualquier caso, este vendrá ya con un método *main* configurado para levantar la ventana principal una vez sea ejecutado.
3. Sin embargo, el proyecto no tiene configurado aún cuál es la clase principal del mismo, lo cual será necesario para poder ejecutar desde el *IDE* la aplicación. Accede a categoría *Run* desde *propiedades del proyecto*, y en *Main clases* selecciona el marco creado en el paso anterior. Guarda los cambios.
4. Ahora podrás abrir en *NetBeans* el fichero con extensión .java creado en el paso 2 tanto en vista código fuente (*Source*) como en vista de diseño (*Design*).

Figura 15.19
Acceso a vista de diseño.

5. Desde la vista de diseño podrás acceder a la paleta de controles para añadir los componentes a tu aplicación con tan solo arrastrar y soltar sobre el elemento contenedor. Automáticamente se irá generando el código asociado para su posicionamiento y agrupamiento.

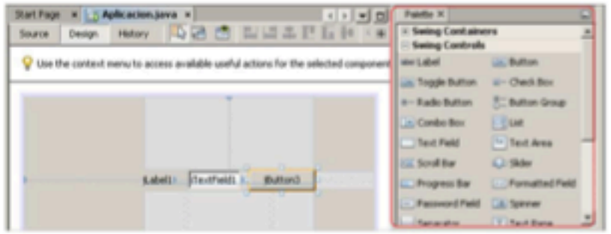
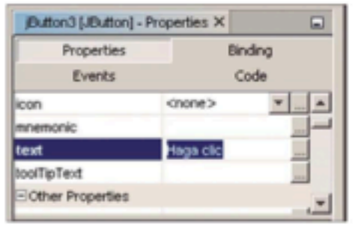


Figura 15.20
Paleta de componentes.

En caso de que no aparezca dicha paleta, podrás activarla desde la opción del menú *Window > IDE Tools > Palette*.

6. Para cada componente podrás establecer sus propiedades, así como fijar los manejadores de eventos. Posteriormente deberás acceder a los métodos manejadores para codificar las acciones necesarias.



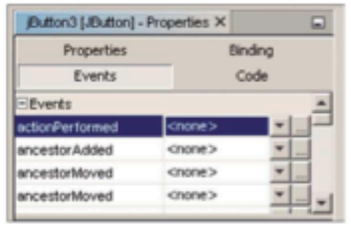


Figura 15.21
Acceso a propiedades y eventos.

Actividad propuesta 15.7

La disposición de los diferentes elementos dentro del IDE es personalizable. Busca cómo posicionar, añadir o eliminar ventanas de este (puedes probar con la ventana de la paleta y la de las propiedades del control mostradas en este apartado).

Resumen

- Las aplicaciones pueden manejarse a través de una interfaz consola o bien a través de interfaces gráficas.
- *AWT* y *Swing* son tecnologías empleadas en el desarrollo de interfaces gráficas en Java.
- El posicionamiento de los controles dentro de un contenedor puede ser absoluto o bien gestionarse mediante un *layout*.
- Los componentes de una interfaz gráfica pueden capturar los eventos que se producen sobre ellos para actuar en consecuencia.
- Los pasos para crear una interfaz gráfica de usuario son:

  1. Crear el área con un contenedor que representará la interfaz de usuario.
  2. Definir el posicionamiento que se realizará con los controles de la interfaz (*layout*).
  3. Crear los componentes que irán dentro del contenedor.
  4. Añadir los componentes al contenedor.
  5. Crear los capturadores de eventos necesarios para cada componente.
  6. Si los componentes creados son a su vez contenedores, repetir con cada uno los pasos desde el punto 1.

Ejercicios propuestos

1. ¿Es importante conocer *AWT* antes de trabajar con *Swing*? Justifica tu respuesta.
2. Realiza una comparativa entre las tecnologías *AWT* y *Swing* en la que evalúes su rendimiento, facilidad de uso y funcionalidad.

3. ¿Existe algún *layout* que permita colocar dos o más componentes en el mismo lugar, pero mostrándose únicamente uno en cada momento?
4. ¿Cuáles son las diferentes regiones de un *BorderLayout*?
5. ¿Hay algún inconveniente en utilizar posicionamiento absoluto en los controles?
6. ¿Es posible cargar nuevos *looks* al sistema para variar el aspecto de las aplicaciones?
7. ¿Puede algún *look* de los mostrados personalizar los colores con los que muestran los componentes?
8. ¿Es posible incluir un control *AWT* en un contenedor *Swing*? ¿Y al contrario?

Supuestos prácticos

1. Implementa un programa calculadora con interfaz gráfica que permita ejecutar las operaciones suma, resta, multiplicación, división y potencia. Utiliza la tecnología que más te convenga para ello.
2. Utilizando la tecnología que quieras, programa una aplicación editora de texto plano. Dispondrás de un área en el que se visualizará y editará el texto, así como un menú con las opciones *Cargar fichero* (para cargar en dicha área el contenido de un fichero) y *Guardar fichero* (que permitirá almacenar en un fichero el contenido del área). Añade a la barra de menú la opción *Reemplazar texto* para permitir la sustitución de una cadena del texto por otra. Por último, añade a la barra de menú la opción *Acerca del editor* para mostrar información sobre el tipo de aplicación, autor, versión…
3. Implementa una aplicación en la que se incorpore una barra de progreso que simule la instalación de un programa. El tiempo de instalación simulado será de 20 segundos.
4. Desarrolla una aplicación que permita gestionar los datos de los alumnos de un colegio. Los datos residirán en una base de datos con la que tendrás que interactuar desde la aplicación. La aplicación deberá permitir la consulta, modificación, inserción y borrado de los datos de los alumnos:
   - Para la inserción se utilizará un formulario que permita introducir: nombre, apellido1, apellido2, DNI, fecha de nacimiento, dirección, curso académico en el que está matriculado y grupo asignado. Además, también deberá permitir almacenar los datos del padre o madre.
   - La aplicación permitirá conocer qué alumnos están en qué grupo y a qué curso pertenece el grupo. Todo ello será mostrado mediante una estructura jerárquica de árbol.
   - Para la consulta de los datos de un alumno estos se mostrarán sobre un formulario similar al empleado para la inserción, pero cuyos campos estarán deshabilitados. Únicamente se habilitarán para proceder con la actualización de los datos.
   - Desde el mismo formulario se podrá realizar también la eliminación de los datos de un alumno.
5. Se desea disponer de un programa que permita formatear el texto de una etiqueta contenida en un formulario a través de diferentes controles. Para ello se insertará en el formulario:

- Un botón que al ser pulsado despliegue una ventana en la que seleccionar el color de la fuente.
- Un conjunto de radiobotones que permitirán indicar si el texto se alinea a la izquierda, centro o derecha.
- Tres *checkbox* para indicar si el texto debe ir en negrita, cursiva o subrayado.

Actividades de autoevaluación

1. ¿Qué método de la clase *Component* permite redimensionar un componente?
   - ☐ *a)* *setSize*.
   - ☐ *b)* *setBounds*.
   - ☐ *c)* Las opciones *a)* y *b)* son correctas.
   - ☐ *d)* Las opciones *a)* y *b)* son incorrectas.
2. ¿Cuál de los siguientes *layout* posiciona los componentes en cinco regiones cardinales?
   - ☐ *a)* *FlowLayout*.
   - ☐ *b)* *BorderLayout*.
   - ☐ *c)* *GridLayout*.
   - ☐ *d)* Todas las opciones anteriores son incorrectas.
3. ¿Cuál de los siguientes contenedores representa una ventana con borde y título que puede incluirse dentro de otro contenedor?
   - ☐ *a)* *Dialog*.
   - ☐ *b)* *Frame*.
   - ☐ *c)* *Panel*.
   - ☐ *d)* Todas las opciones anteriores son incorrectas.
4. Identifica cuál de los siguientes *layout* no está disponible en *AWT*:
   - ☐ *a)* *BorderLayout*.
   - ☐ *b)* *FlowLayout*.
   - ☐ *c)* *GridLayout*.
   - ☐ *d)* *BoxLayout*.
5. Selecciona la opción correcta para TRAILING:
   - ☐ *a)* Los elementos deben estar alineados al comienzo de la región.
   - ☐ *b)* Para el eje vertical la alineación se realiza con el borde inferior.
   - ☐ *c)* Las opciones *a)* y *b)* son correctas.
   - ☐ *d)* Todas las opciones anteriores son incorrectas.

6. ¿Qué tecnología muestra un aspecto u otro en función del sistema operativo anfitrión sobre el cual se ejecute?
   - ☐ *a)* *AWT*.
   - ☐ *b)* *Swing*.
   - ☐ *c)* *JavaFX*.
   - ☐ *d)* Todas las opciones anteriores son incorrectas.
7. ¿Cuál de los siguientes *looks* es utilizado por defecto en una aplicación Java?
   - ☐ *a)* *javax.swing.plaf.nimbus.NimbusLookAndFeel*.
   - ☐ *b)* *com.sun.java.swing.plaf.motif.MotifLookAndFeel*.
   - ☐ *c)* *com.sun.java.swing.plaf.windows.WindowsLookAndFeel*.
   - ☐ *d)* *javax.swing.plaf.metal.MetalLookAndFeel*.
8. Selecciona cuál de los siguientes componentes representa un cuadro de texto que permite al usuario escribir un valor y que contiene dos botones pequeños que posibilitan incrementar o decrementar el valor contenido:
   - ☐ *a)* *JSplitPane*.
   - ☐ *b)* *JSpinner*.
   - ☐ *c)* *JComboBox*.
   - ☐ *d)* *JSeparator*.
9. ¿Qué método retorna la referencia del contenedor en el que se encuentra un control?
   - ☐ *a)* *getParent()*.
   - ☐ *b)* *getParentControl()*.
   - ☐ *c)* *getParentComponentl()*.
   - ☐ *d)* *getContainer()*.
10. ¿Con qué método se configura el comportamiento de un marco?
    - ☐ *a)* *setCloseOperation*.
    - ☐ *b)* *setDefaultExitOperation*.
    - ☐ *c)* *setDefaultClose*.
    - ☐ *d)* *setDefaultCloseOperation*.

SOLUCIONES:

1.	a	b	**c**	d	5.	a	**b**	c	d						
2.	a	**b**	c	d	6.	**a**	b	c	d	9.	**a**	b	c	d	
3.	**a**	b	c	d	7.	a	b	c	**d**	10.	a	b	c	**d**	
4.	a	b	c	**d**	8.	a	**b**	c	d						