



PROGRAMACIÓN

<https://martinezpenya.es/1DAMProgramacion/>

© 2025 David Martínez licensed under CC BY-NC-SA 4.0

1º Programacion (CFGs Desarrollo de Aplicaciones Multiplataforma)

1. IES Eduardo Primo Marques (Carlet)

David Martinez Peña

© 2025 David Martínez licensed under CC BY-NC-SA 4.0

Indice de contenidos

1. UD00	9
1.1 Información importante	9
Denominación del curso	9
Contenidos	9
Resultados de Aprendizaje (RA)	10
Legislación vigente	17
Evaluación	17
2. UD01	19
2.1 Elementos de un programa informático	19
Piensa como un programador	19
Problemas, algoritmos y programas	23
Java	27
Componentes del lenguaje Java	30
Herramientas útiles para empezar	39
Ejemplo UD01	40
Píldoras informáticas relacionadas	40
2.2 Ejercicios de la UD01	41
Retos	41
Ejercicios	41
Expresiones Lógicas	44
Actividades	45
2.3 Talleres	47
Taller UD01_01: Instalar NoMachine para el control remoto	47
Taller UD01_02: Instalación y uso de entornos de desarrollo	48
Taller UD01_03: Crear cuenta en GitHub	73
Taller UD01_T03: Markdown	80
3. UD02	89
3.1 Utilización de Objetos y Clases	89
Introducción a la POO	89
Características de la POO	89
Objetos y Clases	90
Utilización de Objetos	93
Utilización de Métodos	97
Librerías de Objetos (Paquetes)	100
Cadenas de caracteres. La clase <code>String</code>	102

Ejemplo UD02	104
Ejemplos UD02	106
Píldoras informáticas relacionadas	106
3.2 Ejercicios de la UD02	107
Actividades	107
Ejercicios	111
3.3 Talleres	114
Taller UD02_01: GitHub Classroom	114
4. UD03	128
4.1 Estructuras de control y Excepciones	128
Introducción	128
Sentencias y bloques	129
Estructuras de selección	131
Estructuras de repetición	134
Estructuras de salto	139
Excepciones	141
Aserciones (Assertions)	145
Ejemplos UD03	147
Píldoras informáticas relacionadas	155
4.2 Ejercicios de la UD03	156
Retos	156
Ejercicios	158
Actividades	180
4.3 Talleres	185
Taller UD03_01: GitHub Classroom	185
Taller UD03_02: Acepta el reto	186
5. UD04	187
5.1 Estructuras de datos: Arrays y matrices. Recursividad.	187
Introducción	187
Arrays	187
Problemas de recorrido, búsqueda y ordenación	191
Arrays bidimensionales: matrices	196
Arrays multidimensionales	199
Recursividad	199
Ejemplos UD04	204
Píldoras informáticas relacionadas	208
5.2 Anexo Cheatsheet Strings en Java	209
Introducción	209

Construyendo <code>String</code>	209
Operando con Métodos de la clase <code>String</code>	209
Ejemplo de todos los métodos de <code>String</code>	212
Arrays de <code>String</code>	213
Los <code>String</code> son inmutables	213
<code>String</code> en Argumentos de Línea de Comandos	214
Concatenar cadenas en Java	215
5.3 Ejercicios de la UD04	217
Arrays. Ejercicios de recorrido	217
Arrays. Ejercicios de búsqueda	219
Matrices	222
Recursividad	226
5.4 Talleres	228
Taller UD04_01: GitHub Classroom	228
6. UD05	229
6.1 Desarrollo de clases	229
Introducción	230
Estructura y miembros de una clase	231
Atributos	234
Métodos	237
Encapsulación, control de acceso y visibilidad.	244
Utilización de los métodos y atributos de una clase.	245
Constructores.	247
Clases Anidadas, Clases Internas (<i>Inner Class</i>) [NUEVO]	250
Introducción a la herencia. [NUEVO]	253
Conversión entre objetos (Casting) [NUEVO]	254
Acceso a métodos de la superclase [NUEVO]	256
Empaquetado de clases [NUEVO]	256
Ejercicios resueltos	259
Ejemplos UD05	265
Píldoras informáticas relacionadas	271
6.2 Anexo Wrappers y Fechas	272
Wrappers (Envoltorios)	272
Clase <code>Date</code>	274
Ejemplo Anexo UD05	281
6.3 Ejercicios de la UD05	285
Ejercicios	285
Actividades	291

6.4 Talleres	293
Taller UD05_01: GitHub Classroom	293
7. UD06	294
7.1 Lectura y escritura de información	294
Streams (Flujos)	294
Ficheros	297
Serialización	301
Sockets	301
Manejo de ficheros y carpetas (<code>File</code>)	302
Ejemplos UD06	303
Píldoras informáticas relacionadas	313
7.2 Comparativa CRUD	314
7.3 Ejercicios de la UD06	315
Paquetes Completos	315
Los flujos estándar	318
InputStreamReader	319
Entrada "orientada a líneas".	319
Lectura/escritura en ficheros	319
Uso de buffers	319
Streams para información binaria	319
Streams de objetos. Serialización.	320
Sockets	320
Más ejercicios (Lionel)	321
Aún más ejercicios	323
7.4 Talleres	325
Taller UD06_01: GitHub Classroom	325
Taller UD06_T02: Sockets en la nube (AWS)	326
8. UD07	336
8.1 Colecciones	336
Introducción	336
Estructuras de almacenamiento	336
Clases y métodos genéricos	337
Colecciones	340
Iteradores	350
Comparadores	352
Extras	352
Programación funcional	353
Ejemplos UD07	357

Píldoras informáticas relacionadas	369
8.2 Ejercicios de la UD07	370
Ejercicios	370
Actividades	371
Ejercicios Genericidad	374
Programación funcional. Funciones Lambda.	375
8.3 Talleres	377
Taller UD07_01: GitHub Classroom	377
9. UD08	378
9.1 Composición, Herencia y Polimorfismo	378
Relaciones entre clases.	378
Composición	381
Herencia	384
Clases Abstractas	389
Interfaces	392
Polimorfismo	400
Ejemplos UD08	403
Píldoras informáticas relacionadas	426
9.2 Ejercicios de la UD08	427
Ejercicios Herencia	427
Ejercicios Polimorfismo	0
Actividades	0
Ejercicios Lionel	0
9.3 Talleres	0
Taller UD08_01: GitHub Classroom	0
Taller UD08_02: Librerías Maven vs Jar	0
Taller UD08_T03: Introducción a JSON y YAML	0
10. UD09	0
10.1 Interfaz gráfica	0
Introducción	0
Gráfico de escena	0
Controles de la interfaz de usuario	0
Diseño (Layouts)	0
Estructura de la aplicación	0
Píldoras informáticas relacionadas	0
10.2 Ejercicios de la UD09	0
Cuestiones generales	0
Ejercicios (con SceneBuilder)	0

Actividades	0
Más ejercicios (Sin SceneBuilder)	0
10.3 Talleres	0
Taller UD09_01: GitHub Classroom	0
Taller UD09_02: Scene Builder, ScenicView y FXMLManager	0
Taller UD09_03: Proyecto JavaFX (con Maven)	0
Taller UD09_04: Calculadora en JavaFX (IntelliJ)	0
11. UD10	0
11.1 Acceso a Bases de Datos Relacionales	0
Introducción	0
JDBC	0
Patrones de diseño aplicables	0
Acceso a BBDD	0
Navegabilidad y concurrencia	0
Consultas (Query)	0
Modificación (update)	0
Inserción (insert)	0
Borrado (delete)	0
Sentencias predefinidas	0
Píldoras informáticas relacionadas	0
11.2 Ejercicios de la UD10	0
Actividades	0
Para probar...	0
11.3 Talleres	0
Taller UD10_01: GitHub Classroom	0
Taller UD10_02: Conectores	0
Taller UD10_03: BBDD en la nube (AWS) con IntelliJ	0
Taller UD10_04: Patron DAO (CRUD completo)	0
12. UD11	0
12.1 Bases de Datos Orientadas a Objetos	0
Introducción	0
Los lenguajes oDL y oQL	0
La librería ObjectDB	0
12.2 ObjectDB es un potente sistema de gestión de bases de datos orientado a objetos (ODBMS). Es compacto, fiable de usar y extremadamente rápido. ObjectDB proporciona todos los servicios estándar de administración de bases de datos (almacenamiento y recuperación, transacciones, administración de bloqueos, procesamiento de consultas, etc.), pero de una manera que facilita el desarrollo y acelera las aplicaciones.	0
Características clave de la base de datos ObjectDB	0

ObjectDB , que es la BDD elegida permite tanto JDO como JPA (Java Persistence API). Aunque parece que el mercado evoluciona del lado de JPA y es la modalidad más usada para dotar de persistencia a los datos. Por tanto en este tema usaremos ObjectDB con JPA .	0
Anotaciones para ObjectDB	0
Aquí, por su extensión, solo veremos una parte de todo lo que se puede hacer con ObjectDB y JPA, si te interesa profundizar más, te recomiendo que acudas al manual online que tienen publicado en su web, y en el que se basan estos apuntes: https://house.objectdb.com/java/jpa	0
Comparativa BDR vs BDDO vs ORM	0
Píldoras informáticas relacionadas	0
12.3 Ejercicios de la UD11	0
Actividades	0
Ejercicios	0
Ejercicios propuestos	0
Supuestos prácticos	0
Auto evaluación	0
12.4 Talleres	0
Taller UD11_01: GitHub Classroom	0
Taller UD11_02: Añadir ObjectDB a un proyecto IntelliJ (Maven)	0
Taller UD11_03: CRUD con ObjectDB	0
13.  Sobre mí...	0
13.1  David Martínez Peña	0
13.2  Contacto:	0
14. Fuentes de información	0

1. UD00

2. 1.1 Información importante



PROGRAMACIÓN

<https://martinezpenya.es/1DAMProgramacion/>
 © 2025 David Martínez licensed under CC BY-NC-SA 4.0

2.1. Denominación del curso

Ciclo formativo de Grado Superior en Desarrollo de Aplicaciones Multiplataforma

 Programación (PRG)

B5	UD05: Desarrollo de clases	25
B6	UD06: Lectura y escritura de información	25
B4	UD07: Colecciones y Funciones Lambda	

2.2. Contenidos

Bloque	P R I M E R TRIMESTRE	Horas
B1	UD01: Elementos de un programa informático	19
B2	UD02: Utilización de Objetos	20
PRUEBA UNIDADES 1 Y 2		6
B3	UD03: Estructuras de control y Excepciones	20
B4	UD04: Estructuras de datos Arrays y matrices. Recursividad	18
1^a EVALUACIÓN		6
S E G U N D O TRIMESTRE		77
B5	UD05: Desarrollo de clases	25
B6	UD06: Lectura y escritura de información	25
B4	UD07: Colecciones y Funciones Lambda	21
2^a EVALUACIÓN		6
T E R C E R TRIMESTRE		90
B8	UD08: Composición, Herencia y Polimorfismo	20
B9	UD09: Creación de interfaces gráficas	20
B10	UD10: Acceso a Bases de datos	24

		Horas
B11	UD11: BBDD OO	16
	3^a EVALUACIÓN	6
	CONVOCATÒRIA ORDINÀRIA	4
	T O T A L	256

2.3. Resultados de Aprendizaje (RA)

	Descripció	Pes	UNITAT	Avaluació
RA1	Reconoce la estructura de un programa informático, identificando y relacionando los elementos propios del lenguaje de programación utilizado.	10%		
A	Se han identificado los bloques que componen la estructura de un programa informático.	11%	1	= [[1AVA]]
B	Se han creado proyectos de desarrollo de aplicaciones	11%	2	= [[1AVA]]
C	Se han utilizado entornos integrados de desarrollo.	11%	2	= ([[1AVA]]*0,5) + ([[FEE]]*0,5)
D	Se han identificado los distintos tipos de variables y la utilidad específica de cada uno.	11%	1	= ([[1AVA]]*0,2) + ([[2AVA]]*0,3) + ([[3AVA]]*0,5)
E	Se ha modificado el código de un programa para crear y utilizar variables.	11%	1	= ([[1AVA]]*0,2) + ([[2AVA]]*0,3) + ([[3AVA]]*0,5)
F	Se han creado y utilizado constantes y literales.	11%	1	= ([[1AVA]]*0,2) + ([[2AVA]]*0,3) + ([[3AVA]]*0,5)
G	Se han clasificado, reconocido y utilizado en expresiones los operadores del lenguaje.	11%	1	= ([[1AVA]]*0,2) + ([[2AVA]]*0,3) + ([[3AVA]]*0,5)
H	Se ha comprobado el funcionamiento de las conversiones de tipo explícitas e implícitas.	11%	1	= ([[1AVA]]*0,2) + ([[2AVA]]*0,3) + ([[3AVA]]*0,5)
I	Se han introducido comentarios en el código.	11%	1	= ([[1AVA]]*0,2) + ([[2AVA]]*0,3) + ([[3AVA]]*0,5)
	Descripció	Pes	UNITAT	Avaluació
RA2	Escribe y prueba programas sencillos,	10%		

	Descripció	Pes	UNITAT	Avaluació
	reconociendo y aplicando los fundamentos de la programación orientada a objetos.			
A	Se han identificado los fundamentos de la programación orientada a objetos.	11%	2	=[[1AVA]]
B	Se han escrito programas simples.	11%	2	=([[1AVA]]*0,2)+([[2AVA]]*0,3)+([[3AVA]]*0,5)
C	Se han instanciado objetos a partir de clases predefinidas.	11%	2	=([[1AVA]]*0,2)+([[2AVA]]*0,3)+([[3AVA]]*0,5)
D	Se han utilizado métodos y propiedades de los objetos.	11%	2	=([[1AVA]]*0,2)+([[2AVA]]*0,3)+([[3AVA]]*0,5)
E	Se han escrito llamadas a métodos estáticos.	11%	2	=([[1AVA]]*0,2)+([[2AVA]]*0,3)+([[3AVA]]*0,5)
F	Se han utilizado parámetros en la llamada a métodos.	11%	2	=([[1AVA]]*0,2)+([[2AVA]]*0,3)+([[3AVA]]*0,5)
G	Se han incorporado y utilizado librerías de objetos.	11%	2	=([[1AVA]]*0,2)+([[2AVA]]*0,3)+([[3AVA]]*0,5)
H	Se han utilizado constructores.	11%	2	=([[1AVA]]*0,2)+([[2AVA]]*0,3)+([[3AVA]]*0,5)
I	Se ha utilizado el entorno integrado de desarrollo en la creación y compilación de programas simples.	11%	2	=([[1AVA]]*0,05)+([[2AVA]]*0,15)+([[3AVA]]*0,30)+([[FEE]]*0,50)
	Descripció	Pes	UNITAT	Avaluació
RA3	Escribe y depura código, analizando y utilizando las estructuras de control del lenguaje.	10%		
A	Se ha escrito y probado código que haga uso de estructuras de selección.	11%	3	=([[1AVA]]*0,2)+([[2AVA]]*0,3)+([[3AVA]]*0,5)
B	Se han utilizado estructuras de repetición.	11%	3	=([[1AVA]]*0,2)+([[2AVA]]*0,3)+([[3AVA]]*0,5)
C	Se han reconocido las posibilidades de las sentencias de salto.	11%	3	=([[1AVA]]*0,2)+([[2AVA]]*0,3)+([[3AVA]]*0,5)
D	Se ha escrito código utilizando control de excepciones.	11%	3	=([[1AVA]]*0,2)+([[2AVA]]*0,3)+([[3AVA]]*0,5)

	Descripció	Pes	UNITAT	Avaluació
E	Se han creado programas ejecutables utilizando diferentes estructuras de control.	11%	3	=([[1AVA]]*0,2)+([[2AVA]]*0,3)+([[3AVA]]*0,5)
F	Se han probado y depurado los programas.	11%	3	=([[1AVA]]*0,2)+([[2AVA]]*0,3)+([[3AVA]]*0,5)
G	Se ha comentado y documentado el código.	11%	3	=([[1AVA]]*0,05)+([[2AVA]]*0,15)+([[3AVA]]*0,30)+([[FEE]]*0,50)
H	Se han creado excepciones.	11%	3	=([[1AVA]]*0,2)+([[2AVA]]*0,3)+([[3AVA]]*0,5)
I	Se han utilizado aserciones para la detección y corrección de errores durante la fase de desarrollo.	11%	3	=([[1AVA]]*0,2)+([[2AVA]]*0,3)+([[3AVA]]*0,5)
	Descripció	Pes	UNITAT	Avaluació
RA4	Desarrolla programas organizados en clases analizando y aplicando los principios de la programación orientada a objetos.	10%		
A	Se ha reconocido la sintaxis, estructura y componentes típicos de una clase.	11%	5	=([[2AVA]]*0,5)+([[3AVA]]*0,5)
B	Se han definido clases.	11%	5	=([[2AVA]]*0,5)+([[3AVA]]*0,5)
C	Se han definido propiedades y métodos.	11%	5	=([[2AVA]]*0,5)+([[3AVA]]*0,5)
D	Se han creado constructores.	11%	5	=([[2AVA]]*0,5)+([[3AVA]]*0,5)
E	Se han desarrollado programas que instancien y utilicen objetos de las clases creadas anteriormente.	11%	5	=([[2AVA]]*0,5)+([[3AVA]]*0,5)
F	Se han utilizado mecanismos para controlar la visibilidad de las clases y de sus miembros.	11%	5	=([[2AVA]]*0,5)+([[3AVA]]*0,5)
G	Se han definido y utilizado clases heredadas.	11%	5	=([[2AVA]]*0,5)+([[3AVA]]*0,5)
H	Se han creado y utilizado métodos estáticos.	11%	5	=([[2AVA]]*0,5)+([[3AVA]]*0,5)

	Descripció	Pes	UNITAT	Avaluació
I	Se han creado y utilizado conjuntos y librerías de clases.	11%	5	=([[2AVA]]*0,5)+([[3AVA]]*0,5)
	Descripció	Pes	UNITAT	Avaluació
RA5	Realiza operaciones de entrada y salida de información, utilizando procedimientos específicos del lenguaje y librerías de clases.	15%		
A	Se ha utilizado la consola para realizar operaciones de entrada y salida de información.	10%	6	=([[1AVA]]*0,05)+([[2AVA]]*0,15)+([[3AVA]]*0,30)+([[FEE]]*0,50)
B	Se han aplicado formatos en la visualización de la información.	10%	6	=([[1AVA]]*0,2)+([[2AVA]]*0,3)+([[3AVA]]*0,5)
C	Se han reconocido las posibilidades de entrada / salida del lenguaje y las librerías asociadas.	10%	6	=([[2AVA]]*0,5)+([[3AVA]]*0,5)
D	Se han utilizado ficheros para almacenar y recuperar información.	10%	6	=([[2AVA]]*0,5)+([[3AVA]]*0,5)
E	Se han creado programas que utilicen diversos métodos de acceso al contenido de los ficheros.	10%	6	=([[2AVA]]*0,5)+([[3AVA]]*0,5)
F	Se han utilizado las herramientas del entorno de desarrollo para crear interfaces gráficos de usuario simples.	20%	9	'=([[3AVA]]*0,25)+([[FEE]]*0,50)+([[UD09_T01]]*0,25)+([[UD09_T02]]*0,25)+([[UD09_T03]]*0,5)*0,25)
G	Se han programado controladores de eventos.	15%	9	'=([[3AVA]]*0,4)+([[UD09_T01]]*0,25)+([[UD09_T02]]*0,25)+([[UD09_T03]]*0,5)*0,6)
H	Se han escrito programas que utilicen interfaces gráficos para la entrada y salida de información.	15%	9	'=([[3AVA]]*0,4)+([[UD09_T01]]*0,25)+([[UD09_T02]]*0,25)+([[UD09_T03]]*0,5)*0,6)
	Descripció	Pes	UNITAT	Avaluació
RA6	Escribe programas que manipulen información seleccionando y utilizando tipos avanzados de datos.	20%		
A		50%	4	

	Descripció	Pes	UNITAT	Avaluació
	Se han escrito programas que utilicen matrices (arrays) .			=([[2AVA]]*0,5)+([[3AVA]]*0,50)
B	Se han reconocido las librerías de clases relacionadas con tipos de datos avanzados.	5%	7	=[[3AVA]]
C	Se han utilizado listas para almacenar y procesar información.	5%	7	=[[3AVA]]
D	Se han utilizado iteradores para recorrer los elementos de las listas.	5%	7	=[[3AVA]]
E	Se han reconocido las características y ventajas de cada una de la colecciones de datos disponibles.	10%	7	=[[3AVA]]
F	Se han creado clases y métodos genéricos.	5%	7	=[[3AVA]]
G	Se han utilizado expresiones regulares en la búsqueda de patrones en cadenas de texto.	5%	7	=[[3AVA]]
H	Se han identificado las clases relacionadas con el tratamiento de documentos escritos en diferentes lenguajes de intercambio de datos.	5%	7	=([[UD08_T01]]*0,5)+([[UD08_T02]]*0,5)
I	Se han realizado programas que realicen manipulaciones sobre documentos escritos en diferentes lenguajes de intercambio de datos.	5%	7	=([[UD08_T01]]*0,5)+([[UD08_T02]]*0,5)
J	Se han utilizado operaciones agregadas para el manejo de información almacenada en colecciones.	5%	7	=[[3AVA]]
	Descripció	Pes	UNITAT	Avaluació
RA7	Desarrolla programas aplicando características avanzadas de los lenguajes orientados a objetos y del	10%		

	Descripció	Pes	UNITAT	Avaluació
	entorno de programación.			
A	Se han identificado los conceptos de herencia, superclase y subclase.	10%	8	=[[3AVA]]
B	Se han utilizado modificadores para bloquear y forzar la herencia de clases y métodos.	10%	8	=[[3AVA]]
C	Se ha reconocido la incidencia de los constructores en la herencia.	10%	8	=[[3AVA]]
D	Se han creado clases heredadas que sobrescriban la implementación de métodos de la superclase.	10%	8	=[[3AVA]]
E	Se han diseñado y aplicado jerarquías de clases.	10%	8	=[[3AVA]]
F	Se han probado y depurado las jerarquías de clases.	10%	8	=[[3AVA]]
G	Se han realizado programas que implementen y utilicen jerarquías de clases.	10%	8	=[[3AVA]]
H	Se ha comentado y documentado el código.	10%	8	=([[3AVA]]*0,50)+([[FEE]]*0,50)
I	Se han identificado y evaluado los escenarios de uso de interfaces.	10%	8	=[[3AVA]]
J	Se han identificado y evaluado los escenarios de utilización de la herencia y la composición.	10%	8	=[[3AVA]]
	Descripció	Pes	UNITAT	Avaluació
RA8	Utiliza bases de datos orientadas a objetos, analizando sus características y aplicando técnicas para mantener la persistencia de la información.	5%		
A	Se han identificado las características de las bases de datos orientadas a objetos.	13%	11	=([[UD11_T1]]*0,3)+([[UD11_T2]]*0,7)

	Descripció	Pes	UNITAT	Avaluació
B	Se ha analizado su aplicación en el desarrollo de aplicaciones mediante lenguajes orientados a objetos.	13%	11	$=([[UD11_T1]]*0,3)+([[UD11_T2]]*0,7)$
C	Se han instalado sistemas gestores de bases de datos orientados a objetos.	13%	11	$=([[UD11_T1]]*0,3)+([[UD11_T2]]*0,7)$
D	Se han clasificado y analizado los distintos métodos soportados por los sistemas gestores para la gestión de la información almacenada.	13%	11	$=([[UD11_T1]]*0,3)+([[UD11_T2]]*0,7)$
E	Se han creado bases de datos y las estructuras necesarias para el almacenamiento de objetos.	13%	11	$=([[UD11_T1]]*0,3)+([[UD11_T2]]*0,7)$
F	Se han programado aplicaciones que almacenen objetos en las bases de datos creadas.	13%	11	$=([[UD11_T1]]*0,3)+([[UD11_T2]]*0,7)$
G	Se han realizado programas para recuperar, actualizar y eliminar objetos de las bases de datos.	13%	11	$=([[UD11_T1]]*0,3)+([[UD11_T2]]*0,7)$
H	Se han realizado programas para almacenar y gestionar tipos de datos estructurados, compuestos y relacionados.	13%	11	$=([[UD11_T1]]*0,3)+([[UD11_T2]]*0,7)$
	Descripció	Pes	UNITAT	Avaluació
RA9	Gestiona información almacenada en bases de datos relacionales manteniendo la integridad y consistencia de los datos.	10%		
A	Se han identificado las características y métodos de acceso a sistemas gestores de bases de datos relacionales.	14%	10	$=([[UD10_T1]]*0,2)+([[UD10_T2]]*0,3)+([[UD10_T3]]*0,5)$
B	Se han programado conexiones con bases de datos.	14%	10	$=([[UD10_T1]]*0,2)+([[UD10_T2]]*0,3)+([[UD10_T3]]*0,5)$
C		14%	10	

	Descripció	Pes	UNITAT	Avaluació
	Se ha escrito código para almacenar información en bases de datos.			=([[UD10_T1]]*0,2)+([[UD10_T2]]*0,3)+([[UD10_T3]]*0,5)
D	Se han creado programas para recuperar y mostrar información almacenada en bases de datos.	14%	10	=([[UD10_T1]]*0,2)+([[UD10_T2]]*0,3)+([[UD10_T3]]*0,5)
E	Se han efectuado borrados y modificaciones sobre la información almacenada.	14%	10	=([[UD10_T1]]*0,2)+([[UD10_T2]]*0,3)+([[UD10_T3]]*0,5)
F	Se han creado aplicaciones que muestren la información almacenada en bases de datos.	14%	10	=([[UD10_T1]]*0,05)+([[UD10_T2]]*0,15)+([[UD10_T3]]*0,3)+([[FEE]]*0,5)
G	Se han creado aplicaciones para gestionar la información presente en bases de datos relacionales.	14%	10	=([[UD10_T1]]*0,05)+([[UD10_T2]]*0,15)+([[UD10_T3]]*0,3)+([[FEE]]*0,5)

2.4. Legislación vigente

-  [RD-450/2010, BOE 20-05-2010](#) (Antigua ley)
-  [RD 405/2023 29-05-2023](#)
-  [RD 500/2024, BOE 21-05-2024](#)
-  [Curriculum C.V.: ORDE-58/2012, de 5 de setembre \(DOGV núm. 6868, 24.09.2012\)](#) (Antiguo)
-  [Propuesta de Decreto del Consell](#)
-  [Horario](#) (Antigua ley)
-  Horario

2.5. Evaluación

-  La evaluación del módulo se realizará con base en los **Resultados de Aprendizaje (RA)** definidos en el currículo del ciclo formativo de Grado Superior en Desarrollo de Aplicaciones Multiplataforma. Cada RA estará asociado a **criterios de evaluación (CE)** que serán los que determinen el grado de adquisición de las competencias previstas para el módulo.
-  La nota final del módulo se obtendrá a partir de la ponderación de los **RA**, como se mencionó anteriormente. Cada **RA** será evaluado de forma independiente, con calificaciones en una escala de 0 a 10.
-  El alumno debe obtener al menos una nota de **5** en cada **RA** para aprobar el módulo.
-  Si un alumno obtiene menos de un **5** en algún **RA**, tendrá que recuperarlo mediante las actividades/exámenes de recuperación diseñadas específicamente para esos resultados de aprendizaje.
-  En programación los primeros **RA's** se distribuyen entre las 3 evaluaciones, así que tener una buena nota en la primera evaluación no quiere decir que has aprobado los **RA** de esa evaluación.
-  **NUEVO SISTEMA DUAL!!** → Busca tu empresa! 120H (aproximadamente en el mes de mayo, también a partir del 2º trimestre por las mañanas)

 **IMPORTANTE:**

- ! Aprobar las distintas evaluaciones no garantiza aprobar el curso.
- ❤ Puedes aprobar (y con muy buena nota) las dos evaluaciones, tener un RA suspendido y por tanto suspender el módulo.

 15 de enero de 2026

2. UD01

3. 2.1 Elementos de un programa informático



3.1. Piensa como un programador

Una de las acepciones que trae el Diccionario de Real Academia de la Lengua Española (RAE) respecto a la palabra Problema es “**Planteamiento de una situación cuya respuesta desconocida debe obtenerse a través de métodos científicos**”. Con miras a lograr esa respuesta, un problema se puede definir como una situación en la cual se trata de alcanzar una meta y para lograrlo se deben hallar y utilizar unos medios y unas estrategias.

La mayoría de problemas tienen algunos elementos en común: un estado inicial; una meta, lo que se pretende lograr; un conjunto de recursos, lo que está permitido hacer y/o utilizar; y un dominio, el estado actual de conocimientos, habilidades y energía de quien va a resolverlo (Moursund, 1999).

Casi todos los problemas requieren, que quien los resuelve, los divida en submetas que, cuando son dominadas (por lo regular en orden), llevan a alcanzar el objetivo. La solución de problemas también requiere que se realicen operaciones durante el estado inicial y las submetas, actividades (conductuales, cognoscitivas) que alteran la naturaleza de tales estados (Schunk, 1997).

Cada disciplina dispone de estrategias específicas para resolver problemas de su ámbito; por ejemplo, resolver problemas matemáticos implica utilizar estrategias propias de las matemáticas. Sin embargo, algunos psicólogos opinan que es posible utilizar con éxito estrategias generales, útiles para resolver problemas en muchas áreas. A través del tiempo, la humanidad ha utilizado diversas estrategias generales para resolver problemas. Schunk (1997), Woolfolk (1999) y otros, destacan los siguientes métodos o estrategias de tipo general:

- **Ensayo y error** : Consiste en actuar hasta que algo funcione. Puede tomar mucho tiempo y no es seguro que se llegue a una solución. Es una estrategia apropiada cuando las soluciones posibles son pocas y se pueden probar todas, empezando por la que ofrece mayor probabilidad de resolver el problema.

Por ejemplo, una bombilla que no prende: revisar la bombilla, verificar la corriente eléctrica, verificar el interruptor.

- **Iluminación** : Implica la súbita conciencia de una solución que sea viable. Es muy utilizado el modelo de cuatro pasos formulado por Wallas (1921): preparación, incubación, iluminación y verificación.

Estos cuatro momentos también se conocen como proceso creativo. **Algunas investigaciones han determinado que cuando en el periodo de incubación se incluye una interrupción en el trabajo sobre un problema se logran mejores resultados desde el punto de vista de la creatividad.** La incubación ayuda a "olvidar" falsas pistas, mientras que no hacer interrupciones o descansos puede hacer que la persona que trata de encontrar una solución creativa se estanque en estrategias inapropiadas.

Ejemplos:

- Dispones de 6 lapices/palillos/cerillas igual de largos, ¿como puedes formar 4 triángulos iguales y equiláteros?
- Mueve 2 cerillas para seguir teniendo una copa pero con la cereza fuera:



- **Heurística :** Se basa en la utilización de reglas empíricas para llegar a una solución. El método heurístico conocido como "IDEAL", formulado por Bransford y Stein (1984), incluye cinco pasos: Identificar el problema; definir y presentar el problema; explorar las estrategias viables; avanzar en las estrategias; y lograr la solución y volver para evaluar los efectos de las actividades (Bransford & Stein, 1984).

El matemático Polya (1957) también formuló un método heurístico para resolver problemas que se aproxima mucho al ciclo utilizado para programar computadores. A lo largo de esta Guía se utilizará este método propuesto por Polya.

- **Algoritmos** : Consiste en aplicar adecuadamente una serie de pasos detallados que aseguran una solución correcta. Por lo general, cada algoritmo es específico de un dominio del conocimiento. La programación de computadores se apoya en este método.
- **Modelo de procesamiento de información** : El modelo propuesto por Newell y Simon (1972) se basa en plantear varios momentos para un problema (estado inicial, estado final y vías de solución). Las posibles soluciones avanzan por subtemas y requieren que se realicen operaciones en cada uno de ellos.
- **Análisis de medios y fines** : Se funda en la comparación del estado inicial con la meta que se pretende alcanzar para identificar las diferencias.

Luego se establecen submetas y se aplican las operaciones necesarias para alcanzar cada submeta hasta que se alcance la meta global. Con este método se puede proceder en retrospectiva (desde la meta hacia el estado inicial) o en prospectiva (desde el estado inicial hacia la meta).

- **Razonamiento analógico** : Se apoya en el establecimiento de una analogía entre una situación que resulte familiar y la situación problema. Requiere conocimientos suficientes de ambas situaciones.
- **Lluvia de ideas** : Consiste en formular soluciones viables a un problema. El modelo propuesto por Mayer (1992) plantea: definir el problema; generar muchas soluciones (sin evaluarlas); decidir los criterios para estimar las soluciones generadas; y emplear esos criterios para seleccionar la mejor solución. Requiere que los estudiantes no emitan juicios con respecto a las posibles soluciones hasta que terminen de formularlas.
- **Sistemas de producción** : Se basa en la aplicación de una red de secuencias de condición y acción (Anderson, 1990).
- **Pensamiento lateral** : Se apoya en el pensamiento creativo, formulado por Edwar de Bono (1970), el cual difiere completamente del pensamiento lineal (lógico). El pensamiento lateral requiere que se exploren y consideren la mayor cantidad posible de alternativas para solucionar un problema. Su importancia para la educación radica en permitir que el estudiante: explore (escuche y acepte puntos de vista diferentes, busque alternativas); avive (promueva el uso de la fantasía y del humor); libere (use la discontinuidad y escape de ideas preestablecidas); y contrarreste la rigidez (vea las cosas desde diferentes ángulos y evite dogmatismos). Este es un método adecuado cuando el problema que se desea resolver no requiere información adicional, sino un reordenamiento de la información disponible; cuando hay ausencia del problema y es necesario apercibirse de que hay un problema; o cuando se debe reconocer la posibilidad de perfeccionamiento y redefinir esa posibilidad como un problema (De Bono, 1970).

Ejemplos:

- **El dilema del náufrago.** Un náufrago necesita trasladar a su isla de residencia algunos restos del naufragio de su barco, que afloraron en la orilla de la isla de enfrente. Allí tiene un zorro, un conejo y un racimo de zanahorias, que en su bote puede llevar a razón de uno por viaje. ¿Cómo puede llevarlo todo a su isla, sin que el zorro se coma al conejo, ni éste a las zanahorias?

Respuesta: Deberá llevar primero al conejo y dejar al zorro con las zanahorias. Luego volver y llevarse al zorro, que dejará a solas en su isla, tomar al conejo y llevarlo de vuelta a la de enfrente. Despues llevará las zanahorias, dejando al conejo solo y depositándolas junto al zorro. Finalmente regresará para hacer un último viaje con el conejo.

- **El dilema del ascensor.** Un hombre que vive en el décimo piso de un edificio, toma todos los días el ascensor hasta la planta baja, para ir a trabajar. En la tarde, sin embargo, toma de nuevo el mismo ascensor, pero si no hay nadie con él, baja en el séptimo piso y sube el resto de los pisos por la escalera. ¿Por qué?

Respuesta: El hombre es bajito y no logra presionar el botón del décimo piso.

- **La paradoja del globo.** ¿De qué manera podemos pinchar un globo con una aguja, sin que se fugue el aire y sin que el globo estalle?

Respuesta: Debemos pinchar el globo estando desinflado.

- **El dilema del bar.** Un hombre entra a un bar y le pide al barman un vaso de agua. El barman busca debajo de la barra y de golpe apunta al hombre con un arma. Este último da las gracias y se marcha. ¿Qué acaba de ocurrir?

Respuesta: El barman se percató de que el hombre tenía hipo, y decide curárselo dándole un buen susto.

Como se puede apreciar, hay muchas estrategias para solucionar problemas; sin embargo, esta Guía se enfoca principalmente en dos de estas estrategias: Heurística y Algorítmica.

Según Polya (1957), cuando se resuelven problemas, intervienen cuatro operaciones mentales:

1. Entender el problema;
2. Trazar un plan;
3. Ejecutar el plan (resolver);
4. Revisar;

Es importante notar que estas son flexibles y no una simple lista de pasos como a menudo se plantea en muchos de esos textos (Wilson, Fernández & Hadaway, 1993). Cuando estas etapas se siguen como un modelo lineal, resulta contraproducente para cualquier actividad encaminada a resolver problemas.

Es necesario hacer énfasis en la naturaleza dinámica y cíclica de la solución de problemas. En el intento de trazar un plan, los estudiantes pueden concluir que necesitan entender mejor el problema y deben regresar a la etapa anterior; O cuando han trazado un plan y tratan de ejecutarlo, no encuentran cómo hacerlo; entonces, la actividad siguiente puede ser intentar con un nuevo plan o regresar y desarrollar una nueva comprensión del problema (Wilson, Fernández & Hadaway, 1993; Guzdial, 2000).

La mayoría de los textos escolares de matemáticas abordan la Solución de Problemas bajo el enfoque planteado por Polya. Por ejemplo, en "Recreo Matemático 5" (Díaz, 1993) y en "Dominios 5" (Melo, 2001) se pueden identificar las siguientes sugerencias propuestas a los estudiantes para llegar a la solución de un problema matemático:

1. COMPRENDER EL PROBLEMA.

- Leer el problema varias veces
- Establecer los datos del problema (¿marcarlos de alguna manera?)
- Aclarar lo que se va a resolver (¿Cuál es la pregunta?)
- Precisar el resultado que se desea lograr
- Determinar la incógnita del problema
- Organizar la información
- Agrupar los datos en categorías
- Trazar una figura o diagrama.

2. HACER EL PLAN.

- Escoger y decidir las operaciones a efectuar.
- Eliminar los datos inútiles.
- **Descomponer el problema en otros más pequeños.**

3. EJECUTAR EL PLAN (Resolver).

- Ejecutar en detalle cada operación.
- Simplificar antes de calcular.
- Realizar un dibujo o diagrama

4. ANALIZAR LA SOLUCIÓN (Revisar).

- Dar una respuesta completa
- Hallar el mismo resultado de otra manera.
- Verificar por apreciación que la respuesta es adecuada.

Numerosos autores de libros sobre programación, plantean cuatro fases para elaborar un procedimiento que realice una tarea específica. Estas fases concuerdan con las operaciones mentales descritas por Polya para resolver problemas:

1. Analizar el problema (Entender el problema)
2. Diseñar un algoritmo (Trazar un plan)
3. Traducir el algoritmo a un lenguaje de programación (Ejecutar el plan)
4. Depurar el programa (Revisar)

Como se puede apreciar, hay una similitud entre las metodologías propuestas para solucionar problemas matemáticos (Clements & Meredith, 1992; Díaz, 1993; Melo, 2001; NAP, 2004) y las cuatro fases para solucionar problemas específicos de áreas diversas, mediante la programación de computadores.

Problema de la Jirafa

Primera pregunta: ¿Cómo podríamos meter una jirafa dentro de una nevera? Piensa que es un problema para niños y a ellos no se les pasaría por la cabeza trocear al bello animal para resolver un problema.

Segunda pregunta: Repetimos la jugada con distinto protagonista. ¿Cómo metemos un elefante dentro de la nevera?

Tercera pregunta: Imaginemos que el Rey León está celebrando su cumpleaños y ha invitado a todos los animales del reino. Acuden todos excepto uno. ¿Quién falta?

Cuarta pregunta: Estamos frente a un río que debemos cruzar como sea para continuar nuestro camino. El único problema es que esa zona es el hogar de unos cocodrilos muy agresivos y no disponemos de ningún tipo de embarcación para ir al otro lado. ¿Cómo harías para cruzar el río sin morir en el intento?

3.2. Problemas, algoritmos y programas

3.2.1. Problemas

Podríamos decir que la **programación** es una forma de resolución de **problemas**.

Para que un problema pueda resolverse utilizando un programa informático, éste tiene que poder resolverse de forma mecánica, es decir, mediante una secuencia de instrucciones u operaciones que se puedan llevar a cabo de manera **automática** por un ordenador.

Ejemplos de problemas resolubles mediante un ordenador:

- Determinar el producto de dos números a y b.
- Determinar la raíz cuadrada positiva del número 2.
- Determinar la raíz cuadrada positiva de un número n cualquiera.
- Determinar si el número n, entero mayor que uno, es primo.
- Dada la lista de palabras, determinar las palabras repetidas.
- Determinar si la palabra p es del idioma castellano.
- Ordenar y listar alfabéticamente todas las palabras del castellano.
- Dibujar en pantalla un círculo de radio r.
- Separar las silabas de una palabra p.
- A partir de la fotografía de un vehículo, reconocer y leer su matrícula.
- Traducir un texto de castellano a inglés.
- Detectar posibles tumores a partir de imágenes radiográficas.

Por otra parte, el científico Alan Turing, demostró que existen problemas irresolubles, de los que ningún ordenador será capaz de obtener nunca su solución.

Los problemas deben definirse de forma general y precisa, **evitando ambigüedades**.

Ejemplo: Raíz cuadrada.

- Determinar la raíz cuadrada de un número n.
- Determinar la raíz cuadrada de un número n, entero no negativo, cualquiera.

Ejemplo: Dividir.

- Calcular la división de dos números de dos números a y b.
- Calcular el cociente entero de la división a/b, donde a y b son números enteros y b es distinto de cero. ($5/2 = 2$).
- Calcular el cociente real de la división a/b, donde a y b son números reales y b es distinto de cero ($5/2 = 2.5$).

3.2.2. Algoritmos

Dado un problema P, un **algoritmo** es un conjunto de reglas o pasos que indican cómo resolver P en un tiempo finito.

Secuencias de reglas básicas que utilizamos para realizar operaciones aritméticas: sumas, restas, productos y divisiones.

Algoritmo para desayunar

```

1 Begin
2   Sentarse
3   Servirse café con leche
4   Servirse azúcar
5   If tengo tiempo
6     While tengo apetito
7       Untar mantequilla en una tostada
8       Añadir mermelada
9       Comer la tostada
10    End While
11   End If
12   Beberse el café con leche
13   Levantarse
14 End

```

Un algoritmo, por tanto, no es más que la secuencia de pasos que se deben seguir para solucionar un problema específico. La descripción o nivel de detalle de la solución de un problema en términos algorítmicos depende de qué o quién debe entenderlo, interpretarlo y resolverlo.

Los algoritmos son independientes de los lenguajes de programación y de las computadoras donde se ejecutan. Un mismo algoritmo puede ser expresado en diferentes lenguajes de programación y podría ser ejecutado en diferentes dispositivos. Piensa en una receta de cocina, ésta puede ser expresada en castellano, inglés o francés, podría ser cocinada en fogón o vitrocerámica, por un cocinero o más, etc. Pero independientemente de todas estas circunstancias, el plato se preparará siguiendo los mismos pasos.

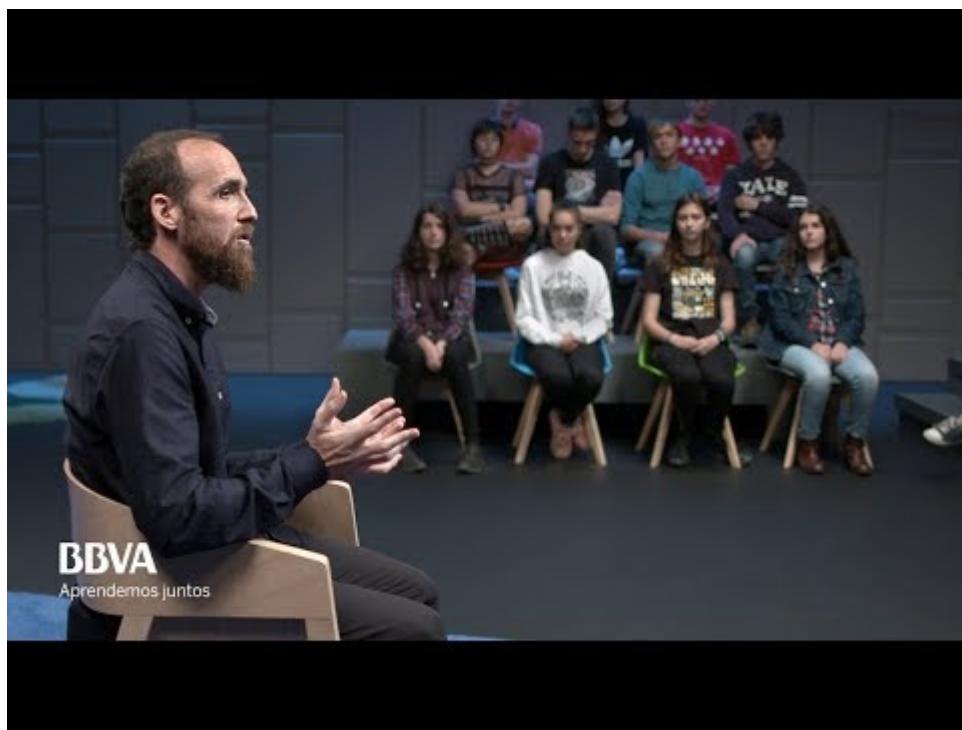
La **diferencia** fundamental entre **algoritmo** y **programa** es que, en el segundo, los pasos que permiten resolver el problema, deben escribirse en un determinado lenguaje de programación para que puedan ser ejecutados en el ordenador y así obtener la solución.

3.2.2.1. CARACTERÍSTICAS DE LOS ALGORITMOS

Un algoritmo, para que sea válido, tiene que tener ciertas características fundamentales:

- **Generalidad:** han de definirse de forma general, utilizando identificadores o parámetros. Un algoritmo debe resolver toda una clase de problemas y no un problema aislado particular.
- **Finitud:** han de llevarse a cabo en un tiempo finito, es decir, el algoritmo ha de acabar necesariamente tras un número finito de pasos.
- **Definibilidad:** han de estar definidos de forma exacta y precisa, sin ambigüedades.
- **Eficiencia:** han de resolver el problema de forma rápida y eficiente.

Juego de las monedas (Eduardo Sáenz Cabezón)



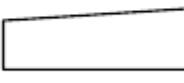
Desde el comienzo del enlace hasta 7 minutos después.

3.2.2.2. REPRESENTACIÓN DE ALGORITMOS

Los métodos más usuales para representar algoritmos son los diagramas de flujo y el pseudocódigo. Ambos son sistemas de representación independientes de cualquier lenguaje de programación. Hay que tener en cuenta que el diseño de un algoritmo constituye un paso previo a la codificación de un programa en un lenguaje de programación determinado (C, C++, Java, Pascal). La independencia del algoritmo del lenguaje de programación facilita, precisamente, la posterior codificación en el lenguaje elegido.

Un **Diagrama de flujo** (Flowchart) es una de las técnicas de representación de algoritmos más antiguas y más utilizadas, aunque su empleo disminuyó considerablemente con los lenguajes de programación estructurados. Un diagrama de flujo utiliza símbolos estándar que contienen los pasos del algoritmo escritos en esos símbolos, unidos por flechas denominadas líneas de flujo que indican la secuencia en que deben ejecutarse.

Los símbolos más utilizados son:

Proceso		Cualquier tipo de operación que pueda originar cambio de valor, formato, operaciones aritméticas etc
Decisión		Indica operaciones lógicas o de comparación entre datos y en función del resultado determina el camino a seguir.
Terminal		Representa el comienzo y el final de un programa o un módulo
Entrada / Salida de información		Este símbolo se puede subdividir en otros de teclado, pantalla, impresora disco etc.
Teclado		Representa las entradas de datos desde teclado
Pantalla		Representa las salidas de datos en pantalla
Dirección del flujo		Indica el sentido de ejecución de las operaciones

Ejemplo: Mostrar dos números ordenados de menor a mayor.

```
graph TD
    A[Inicio] --> B[a, b]
    B --> C{"a > b ?"}
    C -->|Si| D[b, a]
    C -->|No| E[a, b]
    D --> F(Fin)
    E --> F
```

O también en otra representación:



El **pseudocódigo** es un lenguaje de descripción de algoritmos que está muy próximo a la sintaxis que utilizan los lenguajes de programación. Nace como medio para representar las estructuras de control de programación estructurada.

El pseudocódigo no se puede ejecutar nunca en el ordenador, sino que tiene que traducirse a un lenguaje de programación (codificación). La ventaja del pseudocódigo, frente a los diagramas de flujo, es que se puede modificar más fácilmente si detecta un error en la lógica del algoritmo, y puede ser traducido fácilmente a los lenguajes estructurados como Pascal, C, fortran, Java, etc.

El Pseudocódigo utiliza palabras reservadas (en sus orígenes se escribían en inglés) para representar las sucesivas acciones. Para mayor legibilidad utiliza la **indentación** (sangría en el margen izquierdo) de sus líneas.

Ejemplo: Mostrar dos números ordenados de menor a mayor.

```

1 Begin
2   Leer (A, B)
3   If (A>B) then
4     Escribir (B, A)
5   Else
6     Escribir (A, B)
7   End If
8 End
  
```

3.2.3. Programas

La **diferencia** fundamental entre **algoritmo** y **programa** es que, en el segundo, los pasos que permiten resolver el problema, deben escribirse en un determinado lenguaje de programación para que puedan ser ejecutados en el ordenador y así obtener la solución.

Los lenguajes de programación son sólo un medio para expresar el algoritmo y el ordenador un procesador para ejecutarlo. El diseño de los algoritmos será una tarea que necesitará de la creatividad y conocimientos de las técnicas de programación. Estilos distintos, de distintos programadores a la hora de obtener la solución del problema, darán lugar a programas diferentes, igualmente válidos.

Pero cuando los problemas son complejos, es necesario descomponer éstos en subproblemas más simples y, a su vez, en otros más pequeños. Estas estrategias reciben el nombre de diseño descendente (Metodología de diseño de programas, consistente en la descomposición del problema en problemas más sencillos de resolver) o diseño modular (top-down design) (Metodología de diseño de programas, que consiste en dividir la solución a un problema en módulos más pequeños o subprogramas. Las soluciones de los módulos se unirán para obtener la solución general del problema). Este sistema se basa en el lema **divide y vencerás**.

3.3. Java

3.3.1. ¿Qué y cómo es Java?

Java es un lenguaje sencillo de aprender, con una sintaxis parecida a la de C++, pero en la que se han eliminado elementos complicados y que pueden originar errores. Java es orientado a objetos, con lo que elimina muchas preocupaciones al programador y permite la utilización de gran cantidad de bibliotecas ya definidas, evitando reescribir código que ya existe. Es un lenguaje de programación creado para satisfacer nuevas necesidades que los lenguajes existentes hasta el momento no eran capaces de solventar.

Una de las principales virtudes de Java es su independencia del hardware, ya que el código que se genera es válido para cualquier plataforma. Este código será ejecutado sobre una máquina virtual denominada Maquina Virtual Java (MVJ o JVM - Java Virtual Machine), que interpretará el código convirtiéndolo a código específico de la plataforma que lo soporta. De este modo el programa se escribe una única vez y puede hacerse funcionar en cualquier lugar. Lema del lenguaje: "*Write once, run everywhere*".

Antes de que apareciera Java, el lenguaje C era uno de los más extendidos por su versatilidad. Pero cuando los programas escritos en C aumentaban de volumen, su manejo comenzaba a complicarse. Mediante las técnicas de programación estructurada y programación modular se conseguían reducir estas complicaciones, pero no era suficiente.

Fue entonces cuando la Programación Orientada a Objetos (POO) entra en escena, aproximando notablemente la construcción de programas al pensamiento humano y haciendo más sencillo todo el proceso. Los problemas se dividen en objetos que tienen propiedades e interactúan con otros objetos, de este modo, el programador puede centrarse en cada objeto para programar internamente los elementos y funciones que lo componen.

Las características principales de lenguaje Java se resumen a continuación:

- El código generado por el compilador Java es independiente de la arquitectura.
- Está totalmente orientado a objetos.
- Su sintaxis es similar a C y C++.
- Es distribuido, preparado para aplicaciones TCP/IP.
- Dispone de un amplio conjunto de bibliotecas.
- Es robusto, realizando comprobaciones del código en tiempo de compilación y de ejecución.
- La seguridad está garantizada, ya que las aplicaciones Java no acceden a zonas delicadas de memoria o de sistema. (*ejem, ejem!*)

3.3.2. Breve historia.

Java surgió en 1991 cuando un grupo de ingenieros de Sun Microsystems trataron de diseñar un nuevo lenguaje de programación destinado a programar pequeños dispositivos electrónicos. La dificultad de estos dispositivos es que cambian continuamente y para que un programa funcione en el siguiente dispositivo aparecido, hay que reescribir el código. Por eso la empresa Sun quería crear un lenguaje independiente del dispositivo.

Pero no fue hasta 1995 cuando pasó a llamarse Java, dándose a conocer al público como lenguaje de programación para computadores. Java pasa a ser un lenguaje totalmente independiente de la plataforma y a la vez potente y orientado a objetos. Esta filosofía y su facilidad para crear aplicaciones para redes TCP/IP ha hecho que sea uno de los lenguajes más utilizados en la actualidad.

El factor determinante para su expansión fue la incorporación de un intérprete Java en la versión 2.0 del navegador Web Netscape Navigator, lo que supuso una gran revuelo en Internet. A principios de 1997 apareció Java 1.1 que proporcionó sustanciales mejoras al lenguaje. Java 1.2, más tarde rebautizado como Java 2, nació a finales de 1998.

El principal objetivo del lenguaje Java es llegar a ser el nexo universal que conecte a los usuarios con la información, esté ésta situada en el ordenador local, en un servidor Web, en una base de datos o en cualquier otro lugar.

Para el desarrollo de programas en lenguaje Java es necesario utilizar un entorno de desarrollo denominado JDK (Java Development Kit), que provee de un compilador y un entorno de ejecución (JRE - Java RunEnvironment) para los bytecodes generados a partir del código fuente. Al igual que las diferentes versiones del lenguaje han incorporado mejoras, el entorno de desarrollo y ejecución también ha sido mejorado sucesivamente.

Java 2 es la tercera versión del lenguaje, pero es algo más que un lenguaje de programación, incluye los siguientes elementos:

- Un lenguaje de programación: Java.
- Un conjunto de bibliotecas estándar que vienen incluidas en la plataforma y que son necesarias en todo entorno Java. Es el Java Core.
- Un conjunto de herramientas para el desarrollo de programas, como es el compilador de bytecodes, el generador de documentación, un depurador, etc.
- Un entorno de ejecución que en definitiva es una máquina virtual que ejecuta los programas traducidos a bytecodes.

3.3.3. Compilar y ejecutar un programa Java . Uso de la consola.

Veamos los pasos para compilar e interpretar nuestro primer programa escrito en lenguaje Java.

3.3.3.1. ESTRUCTURA Y BLOQUES FUNDAMENTALES DE UN PROGRAMA.

Ejemplo Holamundo.java

```

1  public class Holamundo {
2      // programa Hola Mundo
3      public static void main(String[] args) {
4          /* lo único que hace este programa es mostrar
5             la cadena "Hola Mundo!" por pantalla */
6          System.out.println("Hola Mundo!");
7      }
8 }
```

En Java generalmente una clase lleva el identificador `public` y corresponde con un fichero. El nombre de la clase coincide con el del fichero `.java` respetando mayúsculas y minúsculas.

```

1  public class Holamundo {
2      [...]
3 }
```

El código java en las clases se agrupa en funciones o métodos. Cuando java ejecuta el código de una clase busca la función o método `main()` para ejecutarla. Es público (`public`) estático (`static`) para llamarlo sin instanciar la clase. No devuelve ningún valor (`void`) y admite parámetros (`String[] args`) que en este caso no se han utilizado.

```

1  [...]
2  public static void main (String[] args)
3  {
4      [...]
5  }
6  [...]
```

El código de la función `main` se escribe entre las llaves. Por ejemplo:

```

1  [...]
2      System.out.println("Hola Mundo");
3  [...]
```

Muestra por pantalla el mensaje `Hola Mundo`, ya que la clase `System` tiene un atributo `out` con dos métodos: `print()` y `println()`. La diferencia es que `println` muestra mensaje e introduce un retorno de carro.

Todas las instrucciones menos las llaves `{ }` terminan con punto y coma (`;`).

3.3.3.2. SANGRADO O TABULADO

El sangrado (también conocido como tabulado) deberá aplicarse a toda estructura que esté lógicamente contenida dentro de otra. El sangrado será de un tabulador. **Es suficiente entre 2 y 4 espacios.** Para alguien que empieza a programar suele ser preferible unos 4 espacios, ya que se ve todo más claro.

Las líneas no tendrán en ningún caso demasiados caracteres que impidan que se pueda leer en una pantalla. **Un número máximo recomendable suele estar entre unos 70 y 90 caracteres, incluyendo los espacios de sangrado.** Si una línea debe ocupar más caracteres, tiene que dividirse en dos o más líneas, para ello utiliza los siguientes principios para realizar la división:

- Tras una coma.
- Antes de un operador, que pasará a la línea siguiente.
- Una construcción de alto nivel (por ejemplo, una expresión con paréntesis).
- La nueva línea deberá alinearse con un sangrado lógico, respecto al punto de ruptura

Unos pocos ejemplos, para comprender mejor:

Dividir tras una coma:

```
1  funcion(expresionMuuuyLarga1,
2          expresionMuuuyyyyLarga2,
3          expresionMuuuyyyyLarga3);
```

Mantener la expresión entre paréntesis en la misma línea:

```
1  nombreLargo = nombreLargo2*
2      (nombreLargo3 + nombreLArgo4) +
3      4*nombreLargo5;
```

Siempre hay excepciones. Puede resultar que al aplicar estas reglas, en operaciones muy largas, o expresiones lógicas enormes, el sangrado sea ilegible. En estos casos, el convenio se puede relajar.

3.3.3.3. PASO 1: CREACIÓN DEL CÓDIGO FUENTE

Abrimos un editor de texto (da igual cual sea, siempre que sea capaz de almacenar "texto sin formato" en código ASCII). Una vez abierto escribiremos nuestro primer programa, que mostrará un texto "Hola Mundo" en la consola. De momento no te preocupes si no entiendes lo que escribes, más adelante le daremos sentido. Ahora solo queremos ver si podemos ejecutar java en nuestro equipo.

El código de nuestro programa en Java será el siguiente:

```
1  /* Ejemplo Hola Mundo */
2  public class Ejemplo {
3      public static void main(String[ ] args) {
4          System.out.println("Hola Mundo");
5      }
6  }
```

A continuación guardamos nuestro archivo y le ponemos como nombre `Ejemplo.java`. Debemos seguir una norma dictada por Java, hemos de hacer coincidir nombre del archivo y nombre del programa, tanto en mayúsculas como en minúsculas, y la extensión del archivo habrá de ser siempre `.java`.

```

~ : nano — Konsole
Fitxer Edita Visualitzar Adreces d'interès Arranjament Ajuda
GNU nano 4.8
/* Ejemplo Hola Mundo */
public class Ejemplo {
    public static void main(String[ ] arg) {
        System.out.println("Hola Mundo");
    }
}
^G Ajuda      ^O Desa      ^W On és      ^K Retalla   ^J Justifica ^C Pos Act
^X Surt       ^R Llegeix   ^L Reemplaça ^U Enganxa te^T Ortografia^_ Vés a línia

```

Debemos recordar exactamente la ruta donde guardamos el archivo de ejemplo `Ejemplo.java`.

3.3.3.4. PASO 2: COMPILACIÓN DEL PROGRAMA

Vamos a proceder a compilar e interpretar este pequeño programa Java (no te preocupes si todavía no entiendes el significado de las palabras compilar e interpretar, lo verás en la asignatura de `Entornos de Desarrollo`). Para ello usaremos la consola. Una vez en la consola debemos colocarnos en la ruta donde previamente guardamos el archivo `Ejemplo.java`.

A continuación daremos la instrucción para que se realice **el proceso de compilación del programa**, para lo que escribiremos `javac Ejemplo.java`, donde `javac` es el nombre del compilador (`java compiler`) que transformará el programa que hemos escrito nosotros en lenguaje Java al lenguaje de la máquina virtual Java (`bytecode`), dando como resultado un nuevo archivo `Ejemplo.class` que se creará en este mismo directorio. Comprueba que no aparezca ningún error y que `javac` esté instalado en tu sistema (desde la consola lo puedes comprobar con el comando `javac --version` y debería aparecer el número de versión que tienes instalada). Si aparecen los dos archivos tanto `Ejemplo.java` (código fuente) como `Ejemplo.class` (bytecode creado por el compilador) puedes continuar.

```
1 $ javac Ejemplo.java
```

3.3.3.5. PASO 3: EJECUCIÓN DEL PROGRAMA

Finalmente, vamos a pedirle al intérprete (JVM) que ejecute el programa, es decir, que transforme el código de la máquina virtual Java en código máquina interpretable por nuestro ordenador y lo ejecute. Para ello escribiremos en la ventana consola: `java Ejemplo`.

El resultado será que se nos muestra la cadena `Hola Mundo`. Si logramos visualizar este texto en pantalla, ya hemos desarrollado nuestro primer programa en Java.

```
1 $ java Ejemplo
2 Hola Mundo
```

Por qué no necesito compilar mi archivo `.java` antes de ejecutarlo y funciona directamente si me salto ese paso?

<https://stackoverflow.com/questions/54493058/running-a-java-program-without-compiling>

3.4. Componentes del lenguaje Java

3.4.1. Variables, identificadores, convenciones.

3.4.1.1. VARIABLES

Una **variable** es una zona en la memoria del computador con un valor que puede ser almacenado para ser usado más tarde en el programa. Las variables vienen determinadas por:

- un **nombre**, que permite al programa acceder al valor que contiene en memoria. Debe ser un identificador válido.
- un **tipo de dato**, que especifica qué clase de información guarda la variable en esa zona de memoria
- un **rango de valores** que puede admitir dicha variable.

Las variables declaradas dentro de un bloque `{ }` son accesibles solo dentro de ese bloque. Una variable local no puede ser declarada como `static`. Una variable no puede declararse fuera de la clase.

Visibilidad, ámbito o scope de una variable es la parte de código del programa donde la variable es accesible y utilizable. Las variables de un bloque son visibles y existen dentro de dicho bloque. Las funciones miembro de clase podrán acceder a todas las variables miembro de dicha clase pero no a las variables locales de otra función miembro.

Al nombre que le damos a la variable se le llama identificador. Los identificadores permiten nombrar los elementos que se están manejando en un programa. Vamos a ver con más detalle ciertos aspectos sobre los identificadores que debemos tener en cuenta.

3.4.1.2. IDENTIFICADORES

Un **identificador** en Java es una secuencia ilimitada sin espacios de letras y dígitos Unicode , de forma que el primer símbolo de la secuencia debe ser una letra, un símbolo de subrayado (_) o el símbolo dólar (\$). Por ejemplo, son válidos los siguientes identificadores:

- x5
- ατη
- NUM_MAX
- numCuenta

Unicode es un código de caracteres o sistema de codificación, un alfabeto que recoge los caracteres de prácticamente todos los idiomas importantes del mundo. Además, el código Unicode es “compatible” con el código ASCII, ya que para los caracteres del código ASCII, Unicode asigna como código los mismos 8 bits, a los que les añade a la izquierda otros 8 bits todos a cero. La conversión de un carácter ASCII a Unicode es inmediata.

3.4.1.3. CONVENCIONES

Normas de estilo para nombrar variables

A la hora de nombrar un identificador existen una serie de normas de estilo de uso generalizado que, no siendo obligatorias, se usan en la mayor parte del código Java. Estas reglas para la nomenclatura de variables son las siguientes:

- Java distingue las mayúsculas de las minúsculas. Por ejemplo, `Alumno` y `alumno` son variables diferentes.
- No se suelen utilizar identificadores que comiencen con `$` o `_`, además el símbolo del dólar, por convenio, no se utiliza nunca.
- No se puede utilizar el valor booleano (`true` o `false`) ni el valor nulo (`null`).
- Los identificadores deben ser lo más descriptivos posibles. Es mejor usar palabras completas en vez de abreviaturas crípticas. Así nuestro código será más fácil de leer y comprender. En muchos casos también hará que nuestro código se auto-documente. Por ejemplo, si tenemos que darle el nombre a una variable que almacena los datos de un cliente sería recomendable que la misma se llamara algo así como `FicheroClientes` o `ManejadorCliente`, y no algo poco descriptivo como `c133`.

Además de estas restricciones, en la siguiente tabla puedes ver otras convenciones, que no siendo obligatorias, sí son recomendables a la hora de crear identificadores en Java.

Identificador	Convención	Ejemplo
nombre de variable	Comienza por letra minúscula, y si tienen más de una palabra se colocan juntas y el resto comenzando por mayúsculas. A esto se le llama <i>lowerCamelCase</i> .	numAlumnos, suma
nombre de constante	En letras mayúsculas, separando las palabras con el guión bajo, por convenio el guión bajo no se utiliza en ningún otro sitio	TAM_MAX, PI
nombre de una clase	Comienza por letra mayúscula, y si tienen más de una palabra se colocan juntas y el resto comenzando por mayúsculas. A esto se le llama <i>UpperCamelCase</i> .	String, MiTipo
nombre de función	Comienza por letra minúscula, y si tienen más de una palabra se colocan juntas y el resto comenzando por mayúsculas. A esto se le llama <i>lowerCamelCase</i> .	modificaValor, obtieneValor

Puedes consultar estas y otras convenciones sobre código Java en este [enlace](#).

Palabras reservadas Las palabras reservadas, a veces también llamadas palabras clave o keywords, son secuencias de caracteres formadas con letras ASCII cuyo uso se reserva al lenguaje y, por tanto, no pueden utilizarse para crear identificadores.

Las palabras reservadas en Java son:

```
1 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.
```

3.4.2. Tipos de datos.

Los tipos de datos se utilizan para declarar variables y el compilador sepa de antemano que tipo de información contendrá la variable.

Java dispone de los siguientes tipos de datos simples:

Tipo de dato	Representación	Tamaño (Bytes)	Rango de Valores	Valor por defecto	Clase Asociada
byte	Numérico Entero con signo	1	-128 a 127	0	Byte
short	Numérico Entero con signo	2	-32768 a 32767	0	Short
int	Numérico Entero con signo	4	-2147483648 a 2147483647	0	Integer
long	Numérico Entero con signo	8	-9223372036854775808 a 9223372036854775807	0	Long
float	Numérico en Coma flotante de precisión simple Norma IEEE 754	4	-3.4x10 ⁻³⁸ a 3.4x10 ³⁸	0.0	Float
double	Numérico en Coma flotante de precisión doble Norma IEEE 754	8	-1.8x10 ⁻³⁰⁸ a 1.8x10 ³⁰⁸	0.0	Double
char	Carácter Unicode	2	\u0000 a \uFFFF	\u0000	Character
boolean	Dato lógico	-	true ó false	false	Boolean
void	-	-	-	-	Void

Sobre valores por defecto y inicialización de variables: <https://stackoverflow.com/questions/19131336/default-values-and-initialization-in-java>

Ejemplo de declaración y asignación de valores a variables:

Tipo de datos	código
byte	byte a;
short	short b, c=3;
int	int d=-30; int e=0xC125; //la 0x significa Hexadecimal
long	long b=46240; long b=5L; // La L en este caso indica Long
char	char car1='c'; char car2=99; //car1 y car2 son iguales, la c equivale al ascii 99 char letra = '\u0061'; //código unicode del carácter "a"
float	

Tipo de datos	código
	<pre>float pi=3.1416; float pi=3.1416F; //La F significa float float medio=1/2; //0.5</pre>
double	<pre>double millon=1e6; // 1x10^6 double medio=1/2D; //0.5, la D significa double double z=.123; //si la parte entera es 0 se puede omitir</pre>
boolean	<pre>boolean esPrimero; boolean esPar=false;</pre>

> Ojo con los tipo float: <https://jvns.ca/blog/2023/01/13/examples-of-floating-point-problems/>

3.4.3. Tipos referenciados

A partir de los ocho tipos datos primitivos, se pueden construir otros tipos de datos. Estos tipos de datos se llaman tipos referenciados o referencias, porque se utilizan para almacenar la dirección de los datos en la memoria del ordenador.

```
1 int[] arrayDeEnteros;
2 Cuenta cuentaCliente;
```

En la primera instrucción declaramos una lista de números del mismo tipo, en este caso, enteros. En la segunda instrucción estamos declarando la variable u objeto `cuentaCliente` como una referencia de tipo `Cuenta`.

Cualquier aplicación de hoy en día necesita no perder de vista una cierta cantidad de datos. Cuando el conjunto de datos utilizado tiene características similares se suelen agrupar en estructuras para facilitar el acceso a los mismos, son los llamados datos estructurados.

Son datos estructurados los `arrays`, `listas`, `árboles`, etc. Pueden estar en la memoria del programa en ejecución, guardados en el disco como ficheros, o almacenados en una base de datos.

Además de los ocho tipos de datos primitivos que ya hemos descrito, Java proporciona un tratamiento especial a los textos o cadenas de caracteres mediante el tipo de dato `String`. Java crea automáticamente un nuevo objeto de tipo `String` cuando se encuentra una cadena de caracteres encerrada entre comillas dobles. En realidad se trata de objetos, y por tanto son tipos referenciados, pero se pueden utilizar de forma sencilla como si fueran variables de tipos primitivos:

```
1 String mensaje;
2 mensaje= "El primer programa";
```

Hemos visto qué son las variables, cómo se declaran y los tipos de datos que pueden adoptar. Anteriormente hemos visto un ejemplo de creación de variables, en esta ocasión vamos a crear más variables, pero de distintos tipos primitivos y los vamos a mostrar por pantalla. Los tipos referenciados los veremos en la siguiente unidad.

Para mostrar por pantalla un mensaje utilizamos `System.out`, conocido como la salida estándar del programa. Este método lo que hace es escribir un conjunto de caracteres a través de la línea de comandos. Podemos utilizar `System.out.print` o `System.out.println`. En el segundo caso lo que hace el método es que justo después de escribir el mensaje, sitúa el cursor al principio de la línea siguiente.

El texto en color gris que aparece entre caracteres `//` son comentarios que permiten documentar el código, pero no son tenidos en cuenta por el compilador y, por tanto, no afectan a la ejecución del programa.

3.4.4. Tipos enumerados

Los tipos de datos enumerados son una forma de declarar una variable con un conjunto restringido de valores. Por ejemplo, los días de la semana, las estaciones del año, los meses, etc. Es como si definiéramos nuestro propio tipo de datos.

La forma de declararlos es con la palabra reservada `enum`, seguida del nombre de la variable y la lista de valores que puede tomar entre llaves. A los valores que se colocan dentro de las llaves se les considera como constantes, van separados por comas y deben ser valores únicos.

La lista de valores se coloca entre llaves, porque un tipo de datos `enum` no es otra cosa que una especie de clase en Java, y todas las clases llevan su contenido entre llaves.

Al considerar Java este tipo de datos como si de una clase se tratara, no sólo podemos definir los valores de un tipo enumerado, sino que también podemos definir operaciones a realizar con él y otro tipo de elementos, lo que hace que este tipo de dato sea más versátil y potente que en otros lenguajes de programación.

En el siguiente ejemplo puedes comprobar el uso que se hace de los tipos de datos enumerados.

```

1 public class tiposEnumerados {
2     public enum dias {Lunes, Martes, Miércoles, Jueves, Viernes, Sábado, Domingo};
3
4     public static void main(String[] args) {
5         dias diaActual = dias.Martes;
6         dias diaSiguiente = dias.Miércoles;
7
8         System.out.print("Hoy es: ");
9         System.out.println(diaActual);
10        System.out.println("Mañana\nes\n"+diaSiguiente);
11    }
12 }
```

El resultado después de la ejecución será:

```

1 Hoy es: Martes
2 Mañana
3 es
4 Miércoles
```

Tenemos una variable `Dias` que almacena los días de la semana. Para acceder a cada elemento del tipo enumerado se utiliza el nombre de la variable seguido de un punto y el valor en la lista. Más tarde veremos que podemos añadir métodos y campos o variables en la declaración del tipo enumerado, ya que como hemos comentado un tipo enumerado en Java tiene el mismo tratamiento que las clases.

En este ejemplo hemos utilizado el método `System.out.print`. Como podrás comprobar si lo ejecutas, la instrucción `print` escribe el texto que tiene entre comillas pero no salta a la siguiente línea, por lo que la instrucción `println` escribe justo a continuación.

Sin embargo, también podemos escribir varias líneas usando una única sentencia. Así lo hacemos en la instrucción `println`, la cual imprime como resultado tres líneas de texto. Para ello hemos utilizado un carácter especial, llamado carácter escape (`\`). Este carácter sirve para darle ciertas órdenes al compilador, en lugar de que salga impreso en pantalla. Después del carácter de escape viene otro carácter que indica la orden a realizar, juntos reciben el nombre de secuencia de escape. La secuencia de escape `\n` recibe el nombre de carácter de nueva línea. Cada vez que el compilador se encuentra en un texto ese carácter, el resultado es que mueve el cursor al principio de la línea siguiente. En el próximo apartado vamos a ver algunas de las secuencias de escape más utilizadas.

3.4.5. Constantes y literales.

Las **constantes** se utilizan para almacenar datos que no varían nunca, asegurándonos que el valor no va a poder ser modificado.

Podemos declarar una constante utilizando:

```
1 final <tipo de datos> <nombre de la constante> = <valor>;
```

El calificador `final` indica que es constante. A continuación indicaremos el tipo de dato, el nombre de la constante y el valor que se le asigna.

```
1 final double IVA= 0.21;
```

Los **literales** pueden ser de tipo simple, null o string, como por ejemplo 230, null o "Java".

Respecto a los literales existen unos caracteres especiales que se representan utilizando secuencias de escape:

Secuencia de escape	Significado	Secuencia de escape	Significado
\b	Retroceso	\r	Retorno de carro
\t	Tabulador	\\"	Carácter comillas dobles
\n	Salto de línea	\'	Carácter comillas simples
\f	Salto de página	\\\	Barra diagonal

3.4.6. Operadores y expresiones.

3.4.6.1. OPERADORES ARITMÉTICOS

Los **Operadores Aritméticos** permiten realizar operaciones matemáticas:

Operador	Uso	Operación
+	A + B	Suma
-	A - B	Resta
*	A * B	Multiplicación
/	A / B	División
%	A % B	Módulo o resto de una división entera

Ejemplo:

```

1 double num1, num2, suma, resta, producto, division, resto;
2 num1 =8;
3 num2 =5;
4 suma = num1 + num2;      // 13
5 resta = num1 - num2;     // 3
6 producto = num1 * num2; // 40
7 division = num1 / num2; // 1.6
8 resto = num1 % num2;    // 3

```

3.4.6.2. OPERADORES RELACIONALES

Los **Operadores Relacionales** permiten evaluar (la respuesta es un booleano: si o no) la igualdad de los operandos:

Operador	Uso	Operación
<	a < b	a menor que b
>	a > b	a mayor que b
<=	a <= b	a menor o igual que b
>=	a >= b	a mayor o igual que b
!=	a != b	a distinto de b
==	a == b	a igual a b

Por ejemplo:

```

1 int valor1 = 10;
2 int valor2 = 3;
3 boolean compara;
4 compara = valor1 > valor2; // true
5 compara = valor1 < valor2; // false
6 compara = valor1 >= valor2; // true
7 compara = valor1 <= valor2; // false
8 compara = valor1 == valor2; // false
9 compara = valor1 != valor2; // true

```

3.4.6.3. OPERADORES LÓGICOS

Los **Operadores Lógicos** permiten realizar operaciones lógicas:

Operador	Uso	Operación
&& o &	a&b o a&b	a AND b. El resultado será <i>true</i> si ambos operadores son <i>true</i> y <i>false</i> en caso contrario.
o	a b o a b	a OR b. El resultado será <i>false</i> si ambos operandos son <i>false</i> y <i>true</i> en caso contrario
!	!a	NOT a. Si el operando es <i>true</i> el resultado es <i>false</i> y si el operando es <i>false</i> el resultado es <i>true</i> .

Operador	Uso	Operación
<code>^</code>	<code>a^b</code>	a XOR b. El resultado será <i>true</i> si un operando es <i>true</i> y el otro <i>false</i> , y <i>false</i> en caso contrario.

Ejemplo:

```

1 double sueldo = 1400;
2 int edad = 34;
3 boolean logica;
4 logica = (sueldo<1000 & edad<40); //true
5 logica = (sueldo>1000 && edad >40); //false
6 logica = (sueldo>1000 | edad>40); //true
7 logica = (sueldo<1000 || edad >40); //false
8 logica = !(edad <40); //false
9 logica = (sueldo>1000 ^ edad>40); //true
10 logica = (sueldo<1000 ^ edad>40); //false

```

Para representar resultados de operadores Lógicos también se pueden usar tablas de verdad a las que conviene acostumbrarse:

a	b	<code>a && b</code>	<code>a b</code>	<code>!a</code>	<code>a^b</code>
false	false	false	false	true	false
true	false	false	true	false	true
false	true	false	true	true	true
true	true	true	true	false	false

3.4.6.4. OPERADORES UNARIOS O UNITARIOS

Los **Operadores Unarios o Unitarios** permiten realizar incrementos y decrementos:

Operador	Uso	Operación
<code>++</code>	<code>a++</code> o <code>++a</code>	Incremento de a
<code>--</code>	<code>a--</code> o <code>--a</code>	Decremento de a

Ejemplo:

```

1 int m = 5, n = 3;
2 m++; // 6
3 n--; // 2

```

En el caso de utilizarlo como prefijo el valor de asignación será el valor del operando más el incremento de la unidad. Y si lo utilizamos como sufijo se asignará el valor del operador y luego se incrementará la unidad sobre el operando.

```

1 int a = 1, b;
2 b = ++a; // a vale 2 y b vale 2 //coge lo que vale a, le suma 1 y lo guarda en b
3 b = a++; // a vale 3 y b vale 2 //coge lo que vale a, lo guarda en b, y suma 1 a lo que vale a

```

3.4.6.5. OPERADORES DE ASIGNACIÓN

Los **Operadores de Asignación** permiten asignar valores:

Operador	Uso	Operación
<code>=</code>	<code>a = b</code>	Asignación (como ya hemos visto)
<code>*=</code>	<code>a *= b</code>	Multiplicación y asignación. La operación <code>a*=b</code> equivale a <code>a=a*b</code>
<code>/=</code>	<code>a /= b</code>	División y asignación. La operación <code>a/=b</code> equivale a <code>a=a/b</code>
<code>%=</code>	<code>a %= b</code>	Módulo y asignación. La operación <code>a%=b</code> equivale a <code>a=a%b</code>
<code>+=</code>	<code>a += b</code>	Suma y asignación. La operación <code>a+=b</code> equivale a <code>a=a+b</code>
<code>-=</code>	<code>a -= b</code>	Resta y asignación. La operación <code>a-=b</code> equivale a <code>a=a-b</code>

Ejemplo:

```
1 int dato1 = 10, dato2 = 2, dato;
2 dato=dato1; // dato vale 10
3 dato2*=dato1; // dato2 vale 20
4 dato2/=dato1; // dato2 vale 2
5 dato2+=dato1; // dato2 vale 12
6 dato2-=dato1; // dato2 vale 2
7 dato1%=dato2; // dato1 vale 0
```

3.4.6.6. OPERADORES DE DESPLAZAMIENTO

Los **Operadores de desplazamiento** permiten desplazar los bits de los valores:

Operador	Utilización	Resultado
<code><<</code>	<code>a << b</code>	Desplazamiento de <code>a</code> a la izquierda en <code>b</code> posiciones. Multiplica por 2 el número <code>b</code> de veces.
<code>>></code>	<code>a >> b</code>	Desplazamiento de <code>a</code> a la derecha en <code>b</code> posiciones, tiene en cuenta el signo. Divide por 2 el número <code>b</code> de veces.
<code>>>></code>	<code>a >>> b</code>	Desplazamiento de <code>a</code> a la derecha en <code>b</code> posiciones, no tiene en cuenta el signo. (simplemente agrega ceros por la izquierda)
<code>&</code>	<code>a & b</code>	Operación AND a nivel de bits
<code> </code>	<code>a b</code>	Operación OR a nivel de bits
<code>^</code>	<code>a^b</code>	Operación XOR a nivel de bits
<code>~</code>	<code>~a</code>	Complemento de A a nivel de bits

Por ejemplo:

3.4.6.7. OPERADOR CONDICIONAL O TERNARIO ?:

El operador condicional ?: sirve para evaluar una condición y devolver un resultado en función de si es verdadera o falsa dicha condición. Es el único operador ternario de Java, y como tal, necesita tres operandos para formar una expresión.

El primer operando se sitúa a la izquierda del símbolo de interrogación, y siempre será una expresión booleana, también llamada **condición**. El siguiente operando se sitúa a la derecha del símbolo de interrogación y antes de los dos puntos, y es el **valor** que devolverá el operador condicional **si la condición es verdadera**. El último operando, que aparece después de los dos puntos, es la expresión cuyo **resultado se devolverá si la condición evaluada es falsa**.

1 condición ? exp1 : exp2

Por ejemplo, en la expresión:

```
1 (x>y)?x:y;
```

Se evalúa la condición de si **x es mayor que y**, en caso **afirmativo** se devuelve el valor de la variable **x**, y **en caso contrario** se devuelve el valor de **y**.

Ejemplo para calcular qué número es mayor:

```
1 int mayor, exp1 = 15, exp2 = 25;
2 mayor=(exp1>exp2)?exp1:exp2;
3 // mayor valdrá 25
```

El operador condicional se puede sustituir por la sentencia `if...then...else` que veremos más adelante.

3.4.6.8. PREVALENCIA DE OPERADORES

Los operadores tienen diferente **Prioridad** por lo que es interesante utilizar paréntesis para controlar las operaciones sin necesidad de depender de la prioridad de los operadores.

Prevalencia de operadores, ordenados de arriba a abajo de más a menos prioridad:

Descripción	Operadores
operadores posfijos	op++ op--
operadores unarios	++op --op +op -op ~ !
multiplicación y división	* / %
suma y resta	+ -
desplazamiento	<< >> >>>
operadores relacionales	< > <= =>
equivalencia	== !=
operador AND	&
operador XOR	^
operador OR	
AND booleano	&&
OR booleano	
condicional	? :
operadores de asignación	= += -= *= /= %= &= ^= \ = <=>= >>=

Por ejemplo:

```
1 int x, y1 = 6, y2 = 2, y3 = 8;
2 x = y1 + y2 * y3; // 22
3 x = (y1 + y2) * y3; // 64
```

"Los paréntesis son como las patatas fritas, cuantas más, mejor!" (Ana de mates)

3.4.7. Conversiones de tipo.

Existen dos tipos de conversiones: **Implícitas** y **Explicitas**. Debemos evitar las conversiones de tipos ya que pueden suponer perdidas de información.

3.4.7.1. CONVERSIONES IMPLÍCITAS

Las **Conversiones Implícitas** se realizan de forma automática y requiere que la variable destino tenga más precisión que la variable origen para poder almacenar el valor.

Ejemplo:

```

1 // Conversión Implicita
2 byte origen = 5;
3 short destino;
4 destino=origen; // 5

```

3.4.7.2. CONVERSIÓN EXPLÍCITA

En la **Conversión Explícita** el programador fuerza la conversión con la operación llamada "**cast**":

Ejemplo:

```

1 // Conversión Explicita
2 short origen2 = 3;
3 byte destino2;
4 destino2=(byte)origen2; // 3

```

3.4.8. Comentarios.

Los comentarios son muy importantes a la hora de describir qué hace un determinado programa. A lo largo de la unidad los hemos utilizado para documentar los ejemplos y mejorar la comprensión del código. Para lograr ese objetivo, es normal que cada programa comience con unas líneas de comentario que indiquen, al menos, una breve descripción del programa, el autor del mismo y la última fecha en que se ha modificado.

Todos los lenguajes de programación disponen de alguna forma de introducir comentarios en el código. En el caso de Java, nos podemos encontrar los siguientes tipos de comentarios:

- Comentarios de **una sola línea**. Utilizaremos el delimitador // para introducir comentarios de sólo una línea.

```

1 // comentario de una sola linea
2 byte estoEsUnByte=1;

```

- Comentarios de **múltiples líneas**. Para introducir este tipo de comentarios, utilizaremos una barra inclinada y un asterisco (*), al principio del párrafo y un asterisco seguido de una barra inclinada (/*) al final del mismo.

```

1 /* Esto es un
2 comentario
3 de varias líneas */

```

- Comentarios **Javadoc**. Utilizaremos los delimitadores /** y **/. Al igual que con los comentarios tradicionales, el texto entre estos delimitadores será ignorado por el compilador. Este tipo de comentarios se emplean para generar documentación automática del programa. A través del programa javadoc, incluido en JavaSE, se recogen todos estos comentarios y se llevan a un documento en formato .html.

```

1 /** Comentario de documentación.
2 Javadoc extrae los comentarios del código y
3 genera un archivo html a partir de este tipo de comentarios
4 */

```

3.5. Herramientas útiles para empezar

3.5.1. Generar números aleatorios.

Podemos generar números aleatorios entre 0 y 1 utilizando el método random de la clase `Math`.

```

1 Math.random()

```

Ejemplo:

```

1 double numero;
2 int entero;
3 numero = Math.random();
4 System.out.println("El número es: "+numero); //entre 0 y 0.99999999999999999999999999999999...
5 numero = Math.random()*100;
6 System.out.println("El número es: "+numero); //entre 0 y 99.999999999999999999999999999999...
7 entero = (int)(Math.random()*100);
8 System.out.println("El número sin decimales es: "+entero); //entre 0 y 99
9
10 int lado = ((int)(Math.random()*6))+1;
11 char letra = (char)((Math.random()*26)+65); //65..90
12 System.out.println(letra); //A..Z

```

3.5.2. Introducir un texto desde el teclado.

Este método de leer texto y números desde consola no nos servirá cuando comencemos a usar IDE's.

Podemos introducir texto desde el teclado utilizando `System.console().readLine();`

Ejemplo 1: Introducción de texto.

```

1 String texto;
2 System.out.print("Introduce un texto: ");
3 texto = System.console().readLine();
4 System.out.println("El texto introducido es: "+ texto);

```

Ejemplo 2: Introducción de un número entero.

```

1 String texto2;
2 int entero2;
3 System.out.print("Introduce un número: ");
4 texto2 = System.console().readline();
5 entero2 = Integer.parseInt(texto2); //convertimos texto a Integer
6 System.out.println("El número introducido es:"+entero2);

```

Ejemplo 3: Introducción de un número decimal.

```

1 String texto3;
2 double doble3;
3 System.out.print("Introduce un número decimal: ");
4 texto3 = System.console().readline();
5 doble3 = Double.parseDouble(texto3); // convertimos texto a Double
6 System.out.println("Número decimal introducido es: "+doble3);

```

3.6. Ejemplo UD01

[EjemploUD01.java](#)

3.7. Píldoras informáticas relacionadas

⌚8 de enero de 2026

4. 2.2 Ejercicios de la UD01

4.1. Retos

1. (Reto1) Haga un programa que evalúe una expresión que contenga literales de los cuatro tipos de datos (booleano, entero, real y carácter) y la muestre por pantalla.
2. (Reto2) En su entorno de trabajo, cree el programa siguiente. Obsérvese que pasa exactamente. Entonces, intente arreglar el problema.

```

1 // Un programa que usa un entero muuuuy grande
2 public class TresMilMillones {
3     public static void main (String [] args) {
4         System.out.println (3000000000);
5     }
6 }
```

3. (Reto3) Haga un programa con dos variables que, sin usar ningún literal ninguna parte excepto para inicializar estas variables, vaya estimando e imprimiendo sucesivamente los 5 primeros valores de la tabla de multiplicar del 4. Puede usar operadores aritméticos y de asignación, si desea.
4. (Reto4) Haga dos programas, uno que muestre por pantalla la tabla de multiplicar del 3, y otro, la del 5. Los dos deben ser exactamente iguales, letra por letra, excepto en un único literal dentro de todo el código.
5. (Reto5) Experimente qué pasa si en el siguiente programa inicializa la variable realLargo con un valor con varios decimales. El programa continúa compilando? ¿Qué resultado da? Despues inténtelo asignando un valor superior al rango de los enteros (por ejemplo, 3000000000.0).

```

1 public class ConversionExplicita {
2     public static void main (String[] args) {
3         double realLarg = 300000000.0;
4         // Asignación incorrecta. ¿Un real tiene decimales, no?
5         long enterLlarg = (long) realLlarg;
6         // Asignación incorrecta. ¿Un entero largo tiene un rango mayor que un entero, no?
7         int enter = (int) enterLlarg;
8         System.out.println (enter);
9     }
10 }
```

6. (Reto6) Haga un programa que muestre en pantalla de forma tabulada la tabla de verdad de una expresión de disyunción entre dos variables booleanas.
7. (Reto7) Haga un programa que muestre por pantalla la multiplicación de tres números reales entrados por teclado.

4.2. Ejercicios

Solo se puede usar en esta actividad ya que no se ha explicado en profundidad en este tema y lo pueden confundir con el `System.console().readLine();`

1. (Ejs1) Probar la E/S elemental: Escribe el pequeño programa que aparece a continuación.

```

1 import java.util.*;
2 public class EntradaSalida {
3     public static void main (String arg[]){
4         Scanner tec = new Scanner(System.in);
5         int a, b;
6         System.out.println("Introduce un número entero");
7         a = tec.nextInt();
8         System.out.println("Introduce otro número entero");
9         b = tec.nextInt();
10        System.out.println("Los números introducidos son " + a + " y " + b);
11    }
12 }
```

Ejecútalo para ver como se comporta el programa.

¿Qué ocurre si cuando nos pide un número entero le damos un número real? ¿Y si le damos un carácter no numérico?
¿Qué ocurre si eliminamos la instrucción `import java.util.*;`

2. (Ejs2) Averigua mediante pruebas:

- a. ¿Es posible escribir dos instrucciones en la misma línea de un programa?

- b. ¿Se puede "romper" una instrucción entre varias líneas?
- c. Algunos lenguajes de programación dan un valor por defecto a las variables cuando las declaramos sin inicializarlas. Otros no permiten usar el contenido de una variable que no haya sido previamente inicializada. ¿Cuál es comportamiento de Java?
3. (Esj3) ¿Cuáles de los siguientes identificadores son válidos y cuales no? Pruebalo cuando tengas duda
- n
 - MiProblema
 - MiJuego
 - Mi Juego
 - Int
 - Jose&Co
 - A b
 - 1rApellido
 - aaaaaaaaaaaa
 - NombreApellidos
 - Saldo-actual
 - Universidad Alicante
 - Juan=Rubio
 - Edad5
 - _5Java
 - true
 - _false
 - f_false
4. (Por2) Escribir un programa que lea un entero desde teclado, lo multiplique por 2, y a continuación escriba el resultado en la pantalla:

Ejemplo de ejecución:

```
1 Escribe un número:
2 3
3 El doble de 3 es 6
```

5. (Intercambio) Escribir un programa que ...
- Lea desde teclado dos valores de texto. Llama a las variables s1 y s2.
 - Muestre los valores introducidos por el usuario
 - Intercambie el valor de s1 y s2 (s1 pasa a valer lo que valía s2 y viceversa)
 - Muestre de nuevo los valores, ahora con su valor intercambiado

Ejemplo de ejecución:

```
1 Escribe un texto para s1: David
2 Escribe un texto para s2: Maria
3 Antes de intercambiar    s1: David y    s2: Maria
4 Despues de intercambiar  s1: Maria y    s2: David
```

6. (ExpresionesMatematicas) Escribir las siguientes expresiones siguiendo la sintaxis de Java.
- $\frac{x}{y} + 1$
 - $\frac{x+y}{x-y}$
 - $\left[\frac{b}{c+d} \right]$
 - $(a+b)^2$
 - $x + \frac{y}{2} - \frac{y}{z}$
 - $\frac{xy}{1-4zx}$
 - $\frac{(a+b)c}{d}$
 - $\frac{xy}{mn}$
7. (Superficie) Escribir un programa que solicite al usuario la longitud y la anchura de una habitación y a continuación muestre su superficie (longitud por anchura).

8. (Medidas) Escribir un programa que convierta una medida dada en pies a sus equivalentes en yardas, pulgadas, centímetros y metros, sabiendo que 1 pie = 12 pulgadas, 1 yarda = 3 pies, 1 pulgada = 2.54 cm, 1 m = 100 cm.
9. (Segundos) Escribir un programa que, dada una cantidad de segundos, introducida por teclado, la desglose en días, horas, minutos y segundos.

Ejemplo de ejecución:

```
1 Introduce cantidad de segundos: 3661
2 3661 segundos son:
3 0 dias
4 1 horas
5 1 minutos
6 1 segundos
```

10. (Fuerza) La fuerza de atracción entre dos masas m_1 y m_2 separadas por una distancia d , está dada por la fórmula: $\frac{G \cdot m_1 \cdot m_2}{d^2}$
donde G es la constante de gravitación universal $G = 6.67430 \cdot 10^{-11}$.
- Escribir un programa que lea la masa de dos cuerpos y la distancia entre ellos y a continuación obtenga su fuerza de atracción.
11. (Círculo) Escribir un programa que calcule la longitud de la circunferencia y el área del círculo para un valor del radio introducido por teclado.
12. (Dados) Escribir un programa que simula el lanzamiento de dos dados.

```
1 Dado 1 : 5
2 Dado 2: 4
3 Puntuación total: 9
```

13. (UltimaCifra) Escribir un programa que muestre la última cifra de un número entero que introduce el usuario por teclado.

Pista: ¿Qué devuelve $a \% 10$?

```
1 Introduce un número entero: 3761
2 La última cifra de 3761 es 1
```

14. (PenultimaCifra) Escribir un programa que muestre la penúltima cifra de un número entero que introduce el usuario por teclado.

```
1 Introduce un número entero: 3761
2 La penúltima cifra de 3761 es 6
```

Una vez hayas comprobado que el programa funciona correctamente, prueba qué ocurre si el usuario introduce un valor de una sola cifra (por ejemplo 4). Explica el resultado mostrado por el programa.

15. (Redondear1) `Math.round(x)` redondea x de manera que este queda sin decimales. (`Math.round(35.5289)` da como resultado 36)
- Trata de escribir un programa en el que el usuario introduzca un número real y a continuación se muestre redondeado a un decimal. *Pista : combinar productos, divisiones y Math.round()*

Ejemplo de ejecución:

```
1 Introduce un número real: 35.5289
2 El número 35.5289, redondeado a un decimal es 35.5
```

16. (ExpresionesAritmeticas) 16. Cuál es el valor resultante de dada una de las siguientes expresiones

- a. $5 * 4 - 3 * 6$
- b. $4 * 5 * 2$
- c. $(24 + 2 * 6) / 4$
- d. $8 / 2 / 2 * 5$
- e. $3 + 4 * (8 * (4 - (9 + 3) / 6))$
- f. $4 * 3 * 5 + 8 * 4 * 2$
- g. $4 - 40 \% 5$
- h. $4 * 3 / 2$
- i. $4 / 2 * 3$
- j. $213 / 100$

17. (Einstein) La famosa ecuación de Einstein para la conversión de una masa m en energía viene dada por la fórmula $E=mc^2$, donde c es la velocidad de la luz que vale $2.997925 \cdot 10^8$ m/s. Escribir un programa que lea el valor de la masa y obtenga la energía correspondiente según la anterior fórmula.
18. (FragmentosCódigo) Indica cuales serán los valores de las variables después de ejecutar cada uno de los siguientes fragmentos de código. Resuelve el ejercicio sin escribir los programas correspondientes y probarlos.
- `java int a=3, b = 2; a = b + b; b = a + a;`
 - `java int a=3,b=0; b = b - 1; a = a + b;`
 - `java int a, b=5; b++; ++b; a= b+1;`
 - `java int a = 5,b; b = a++;`
 - `java int a = 5,b; b = ++a;`
 - `java int a=2, b=3; b+=a;`
 - `java int a=2, b=3; b-=a; a=-b;`
 - `java int a=2, b=3; b%=a;`
 - `java int a=2,b=3,c=4; a = --b + c++; b+=a;`

4.3. Expresiones Lógicas

1. Sean 4 variables enteras:

```
1  int m, j, p, v ;
```

que contienen respectivamente la edad de Miguel, Julio, Pablo y Vicente.

Expresar las siguientes afirmaciones utilizando operadores lógicos y relacionales

Ejemplo: Miguel es mayor de edad.

Solución: $m \geq 18$

- (Logica1) Miguel es menor de edad.
- (Logica2) Miguel es mayor que Julio
- (Logica3) Miguel es el más viejo.
- (Logica4) Miguel es el más joven.
- (Logica5) Miguel no es el más joven.
- (Logica6) Miguel no es el más viejo.
- (Logica7) Alguno de ellos es mayor de edad.
- (Logica8) Miguel y Julio son los más jóvenes.
- (Logica9) Entre todos tienen más de 100 años.
- (Logica10) Entre Miguel y Julio suman más edad que Pablo.
- (Logica11) Entre Miguel y Julio suman más edad que Pablo y Vicente juntos.
- (Logica12) Si los ordenamos por edades de menor a mayor, Julio es el segundo.
- (Logica13) Si los ordenamos por edades de menor a mayor, Julio es el segundo y Pablo el tercero.
- (Logica14) Al menos uno de ellos es menor de edad.
- (Logica15) Al menos dos de ellos son menores de edad.
- (Logica16) Todos son menores de edad.
- (Logica17) Solo dos de ellos son menores de edad.
- (Logica18) Al menos dos de ellos nacieron el mismo año.
- (Logica19) Solo dos de ellos nacieron el mismo año.
- (Logica20) Al menos uno de ellos es menor que Julio
- (Logica21) Solo uno de ellos es menor que Julio
- (Logica22) Miguel es mayor de edad y alguno de los otros es menor de edad.

4.4. Actividades

1. (Actividad1) Realiza un conversor de euros a pesetas. La cantidad de euros que se quiere convertir debe ser introducida por teclado.
2. (Actividad2) Realiza un conversor de pesetas a euros. La cantidad de pesetas que se quiere convertir debe ser introducida por teclado.
3. (Actividad3) Escribe un programa que calcule el área de un rectángulo. (`area = base * altura`)
4. (Actividad4) Escribe un programa que calcule el área de un triángulo. (`area = (base * altura) / 2`)
5. (Actividad5) Escribe un programa que calcule el salario semanal de un empleado en base a las horas trabajadas, a razón de 12 euros la hora.
6. (Actividad6) Realiza un conversor de MiB a KiB. [Ayuda](#)
7. (Actividad7) Realiza un conversor de Kib a Mib. [Ayuda](#)
8. (Actividad8) Realiza un programa en Java que genere letras de forma aleatoria.
9. (Actividad9) Realiza un programa en Java que genere el número premiado del Cupón de la ONCE.
10. (Actividad10) Modificar el siguiente programa para que compile y funcione:

```

1  public class Activ10 {
2      public static void main(String[] args) {
3          int n1 = 50, int n2 = 30,
4              boolean suma = 0;
5          suma = n1 + n2;
6          System.out.println("LA SUMA ES: " + suma);
7      }
8  }
```

11. (Actividad11) Modificar el siguiente programa para que compile y funcione:

```

1  public class Activ11 {
2      public static void main(String[] args) {
3          int numero = 2;
4          cuad = numero * numero;
5          System.out.println("EL CUADRADO DE "+NUMERO+" ES: "+cuad);
6      }
7  }
```

12. (Actividad12) Indicar que valor devolverá la ejecución del siguiente programa:

```

1  public class Activ12 {
2      public static void main(String[] args) {
3          int num = 5;
4          num += num - 1 * 4 + 1;
5          System.out.println(num);
6      }
7  }
```

13. (Actividad13) Indicar que valor devolverá la ejecución del siguiente programa:

```

1  public class Activ13 {
2      public static void main(String[] args) {
3          int num = 4;
4          num %= 7 * num % 3 * 3;
5          System.out.println(num);
6      }
7  }
```

14. (Actividad14) Realizar un programa que muestre por pantalla respetando los saltos de carro el siguiente texto (con un solo `println`):

```

1  Me gusta la programación
2  cada día más
```

15. (Actividad15) Realiza un programa en Java que tenga las variables edad, nivel de estudios e ingresos y almacene en una variable llamada jasp el valor verdadero si la edad es menor o igual a 28 y el nivel de estudios es mayor a 3, o bien la edad es menor de 30 y los ingresos superiores a 28000. En caso contrario almacenar el valor falso.
16. (Actividad16) Realizar un programa que realice el cálculo del precio de un producto teniendo en cuenta que el producto vale 120 €, tiene un descuento del 15% y el IVA que se le aplica es del 21%.

17. (Actividad17) Realiza un programa que calcule la nota que hace falta sacar en el segundo examen de la asignatura Programación para obtener la media deseada. Hay que tener en cuenta que la nota del primer examen cuenta el 40% y la del segundo examen un 60%. Ejemplo 1:

```
1 Introduce la nota del primer examen: 7
2 ¿Qué nota quieras sacar en el trimestre? 8.5
3 Para tener un 8.5 en el trimestre necesitas sacar un 9.5 en el segundo examen.
```

Ejemplo 2:

```
1 Introduce la nota del primer examen: 8
2 ¿Qué nota quieras sacar en el trimestre? 7
3 Para tener un 7 en el trimestre necesitas sacar un 6.333333333 en el segundo examen.
```

18. (Actividad18) Realizar un programa que dado un importe en euros nos indique el mínimo número de billetes y la cantidad sobrante de euros. Debes usar el operador condicional ?:

```
1 ¿Cuántos euros tienes?: 232
2 1 billete de 200 €
3 1 billete de 20 €
4 1 billete de 10 €
5 Sobran 2 €
```

⌚5 de octubre de 2025

5. 2.3 Talleres

5.1. Taller UD01_01: Instalar NoMachine para el control remoto

5.1.1. ¿Qué es NoMachine ?

Conéctese a cualquier computadora de forma remota a la velocidad de la luz. Gracias a nuestra tecnología NX, NoMachine es el escritorio remoto más rápido y de mayor calidad que jamás haya probado. Conecta con tu ordenador al otro lado del mundo con solo unos pocos clics. Vé donde esté tu escritorio, podrás acceder a él desde cualquier otro dispositivo y compartirlo con quien quieras. NoMachine es tu servidor personal, privado y seguro. Además, es gratis.

<https://www.nomachine.com/>

5.1.1.1. DESCARGA E INSTALA LA APLICACIÓN

Desde la página de descargas:

<https://downloads.nomachine.com/>

E instala la aplicación en tu PC.

5.1.2. Permisos al profesor

Debemos conceder permisos para que el profesor se pueda conectar a nuestro PC mientras estemos en el instituto sin necesidad de contraseña. Esto solo será posible cuando estemos conectados a la red del instituto, y el profesor no podrá acceder cuando estemos en casa.

Agrega la clave SSH pública (es un fichero `authorized.crt` que te proporcionará el profesor a través de AULES) en tu ordenador

- Debes colocarla en la carpeta `<Inicio del usuario>/.nx/config`.
- Cree este directorio si no existe.
- En Linux y macOS, ejecute en una terminal: `mkdir $HOME/.nx/config`
- En Windows, créelo en (`C:\Users\username\.nx\config`) usando las herramientas del sistema (Explorador de archivos).
- Si la carpeta de configuración ya existe, copia el fichero `authorized.crt` en ella.

Ten en cuenta que los navegadores pueden cambiar las extensiones de los archivos, es conveniente tener las opciones de "ver extensiones de archivos" y "ver archivos ocultos" en nuestro gestor de archivos habitual

5.1.3. Tarea

Debes enviar un archivo `*.pdf` a la plataforma de AULES con una simple captura que demuestre que el profesor se ha podido conectar a tu PC.

Debes mantener `NoMachine` instalado y permitir las conexiones automáticas por parte del profesor para pedir ayuda y consultar dudas en clase, para corregir las tareas diarias, y para realizar los exámenes.

⌚26 de septiembre de 2025

5.2. Taller UD01_02: Instalación y uso de entornos de desarrollo

5.2.1. Java

Cada software y cada entorno de desarrollo tiene unas características y funcionalidades específicas. Esto también se verá reflejado en la instalación y configuración del software. Dependiendo de la plataforma, entorno o sistema operativo en el que se vaya a instalar el software, se utilizará un paquete de instalación u otro, y habrá que tener en cuenta unas opciones u otras en su configuración. A continuación se muestra cómo instalar una herramienta de desarrollo de software integrada, como Eclipse. Pero también podrás observar los procedimientos para instalar otras herramientas necesarias o recomendadas para trabajar con el lenguaje de programación JAVA, como Tomcat o la Máquina Virtual de Java. Debes tener en cuenta los siguientes conceptos:

- La JVM (Java Virtual Machine, máquina virtual de Java) es la encargada de interpretar el bytecode y generar el código máquina del ordenador (o dispositivo) en el que se ejecuta la aplicación. Esto quiere decir que necesitamos una JVM distinta para cada entorno.
- JRE (Java Runtime Environment) es un conjunto de utilidades Java que incluye la JVM, las bibliotecas y el conjunto de software necesario para ejecutar aplicaciones cliente Java, así como el conector para que los navegadores de Internet ejecuten applets.
- JDK (Java Development Kit) es el conjunto de herramientas para desarrolladores; contiene, entre otras cosas, el JRE y el conjunto de herramientas necesarias para compilar el código, empaquetarlo, generar documentación...

```
graph TD
    subgraph JDK
        subgraph JRE
            subgraph JVM
                end
            end
        end
    end
```

El proceso de instalación consta de los siguientes pasos: 1. Descargue, instale y configure el JDK. 2. Descargue e instale un servidor web o de aplicaciones. 3. Descargue, instale y configure el IDE (Netbeans o Eclipse). 4. Configurar JDK con IDE. 5. Configure el servidor web o de aplicaciones con el IDE instalado. 6. Si es necesario, instalación de conectores. 7. Si es necesario, instale un nuevo software.

5.2.1.1. DESCARGUE E INSTALE EL JDK

Podemos diferenciar entre:

- Java SE (Java Standard Edition): es la versión estándar de la plataforma, siendo esta plataforma la base para todos los entornos de desarrollo Java ya sea de aplicaciones cliente, de escritorio o web.
- Java EE (Java Enterprise Edition): esta es la versión más grande de Java y generalmente se utiliza para crear grandes aplicaciones cliente/servidor y para el desarrollo de servicios web.

En este curso se utilizarán las funcionalidades de Java SE. El archivo es diferente según el sistema operativo donde se tenga que instalar. Así:

- Para los sistemas operativos Windows y Mac OS hay un archivo instalable.
- Para los sistemas operativos GNU/Linux que admiten paquetes .rpm o .deb, también están disponibles paquetes de este tipo.
- Para el resto de sistemas operativos GNU/Linux existe un archivo comprimido (terminado en .tar.gz).

En los dos primeros casos, simplemente hay que seguir el procedimiento de instalación habitual del sistema operativo con el que estamos trabajando. En este último caso, sin embargo, hay que descomprimir el archivo y copiarlo en la carpeta donde se desea instalar. Normalmente, todos los usuarios tendrán permisos de lectura y ejecución en esta carpeta.

A partir de la versión 11 de JDK, Oracle distribuye el software con una licencia significativamente más restrictiva que las versiones anteriores. En particular, solo se puede utilizar para "desarrollar, probar, crear prototipos y demostrar sus aplicaciones". Cualquier uso "para fines comerciales, de producción o empresariales internos" distinto del mencionado anteriormente queda explícitamente excluido.

Si lo necesitas para alguno de estos usos no permitidos en la nueva licencia, además de las versiones anteriores del JDK, existen versiones de referencia de estas versiones licenciadas "GNU General Public License version 2, with the Classpath Exception", que permiten la mayoría de los usos habituales. Estas versiones están enlazadas a la misma página de descarga y también a la dirección jdk.java.net.

Una alternativa es utilizar <https://adoptium.net/> antes conocido como adoptOpenJDK, que ahora se ha integrado en la fundación Eclipse. Desde allí podemos descargar los binarios de la versión openJDK para nuestra plataforma sin restricciones. [Noticia completa] (<https://es.wikipedia.org/wiki/OpenJDK>).

En GNU/Linux podemos utilizar los comandos:

- `sudo apt install default-jdk` para instalar el jdk predeterminado.
- `java --version` para ver las versiones disponibles en nuestro sistema.
- `sudo update-alternatives --config java` para elegir cuál de las versiones instaladas queremos usar por defecto o incluso ver la ruta de las diferentes versiones que tenemos instaladas.

5.2.1.2. CONFIGURAR LAS VARIABLES DE ENTORNO "JAVA_HOME" Y "PATH"

Una vez descargado e instalado el JDK, debes configurar algunas variables de entorno:

- La variable `JAVA_HOME`: indica la carpeta donde se ha instalado el JDK. No es obligatorio definirla, pero es muy cómodo hacerlo, ya que muchos programas buscan en ella la ubicación del JDK. Además, resulta muy fácil definir las dos variables siguientes.
- La variable `PATH`. Debe apuntar al directorio que contiene el ejecutable de la máquina virtual. Suele ser la subcarpeta `bin` del directorio donde hemos instalado el JDK.

Variable CLASSPATH Otra variable que tiene en cuenta el JDK es la variable `CLASSPATH`, que apunta a las carpetas donde se encuentran las librerías de la aplicación que se quiere ejecutar con el comando `java`. Es preferible, no obstante, indicar la ubicación de estas carpetas con la opción `-cp` del mismo comando `java`, ya que cada aplicación puede tener diferentes librerías y las variables de entorno afectan a todo el sistema. Establecer la variable `PATH` es esencial para que el sistema operativo encuentre los comandos JDK y pueda ejecutarlos.

5.2.2. Eclipse

Eclipse es una aplicación de código abierto desarrollada actualmente por Eclipse Foundation, una organización independiente, sin fines de lucro, que fomenta una comunidad de código abierto y el uso de un conjunto de productos, servicios, capacidades y complementos para la divulgación del uso de código abierto en el desarrollo de aplicaciones informáticas. Eclipse fue desarrollado originalmente por IBM como sucesor de VisualAge. Como Eclipse está desarrollado en Java, es necesario, para su ejecución, tener un JRE (Java Runtime Environment) previamente instalado en el sistema. Para saber si tienes este JRE instalado, puedes hacer el test en la web oficial de Java, en la sección ¿Tengo Java? Si vamos a desarrollar con Java, como es nuestro caso, deberemos tener instalado el JDK (recordemos que es un superconjunto del JRE).

5.2.2.1. INSTALACIÓN

Las versiones actuales del entorno Eclipse se instalan con un instalador. Este, básicamente, se encarga de descomprimir, solucionar algunas dependencias y crear los accesos directos. Este instalador se puede obtener descargándolo directamente desde la página oficial del Proyecto Eclipse www.eclipse.org. Podrás encontrar las versiones para los diferentes sistemas operativos e instrucciones para su uso. No son nada complejas. En el caso de GNU/Linux y MAC OS, el archivo es un archivo comprimido, por lo que hay que descomprimirlo y luego ejecutar el instalador. Se trata del archivo `eclipse-inst`, dentro de la carpeta `eclipse`, que es una subcarpeta del resultado de descomprimir el archivo anterior. Si sólo el usuario actual va a utilizar el IDE, la instalación se puede realizar sin utilizar privilegios de administrador o root y seleccionando para la instalación una carpeta perteneciente a este usuario. Si se desea compartir la instalación entre distintos usuarios, se debe indicar al instalador una carpeta sobre la que todos estos usuarios tengan permisos de lectura y ejecución.

Al iniciar el instalador veremos una pantalla similar a esta:



El instalador nos preguntará qué versión queremos instalar. La versión que utilizaremos es "Eclipse IDE for Java EE Developers".

A screenshot of the Eclipse Installer interface showing three available IDE options: "Eclipse IDE for Java Developers", "Eclipse IDE for Enterprise Java and Web Developers", and "Eclipse IDE for C/C++ Developers".

- Eclipse IDE for Java Developers**
The essential tools for any Java developer, including a Java IDE, a Git client, XML Editor, Maven and Gradle integration.
- Eclipse IDE for Enterprise Java and Web Developers**
Tools for developers working with Java and Web applications, including a Java IDE, tools for JavaScript, TypeScript, JavaServer Pages and Faces, Yaml, Markdown, Web Services, JPA and...
- Eclipse IDE for C/C++ Developers**
An IDE for C/C++ developers.

Luego nos pedirá la versión de JDK/JRE que vamos a utilizar (en la captura aparece con letras blancas). También nos pide la carpeta donde la instalaremos. Y dos check boxes para indicar si queremos que nos cree el acceso directo al menú de aplicaciones ya en el escritorio.

The screenshot shows the Eclipse Installer interface for the Java EE IDE. At the top, there's a navigation bar with a 'DONATE' button and a gear icon with an exclamation mark. The main title is 'eclipseinstaller by Oomph'. Below the title, there's a circular icon with a gear and the text 'Java EE IDE'.

Eclipse IDE for Enterprise Java and Web Developers [details](#)

Tools for developers working with Java and Web applications, including a Java IDE, tools for JavaScript, TypeScript, JavaServer Pages and Faces, Yaml, Markdown, Web Services, JPA and Data Tools, Maven and Gradle, Git, and more.

Java 11+ VM: /usr/lib/jvm/java-1.11.0-openjdk-amd64

Installation Folder: /home/ubuntu/eclipse/jee-2021-06 [/usr/lib/jvm/java-1.11.0-openjdk-amd64](#)

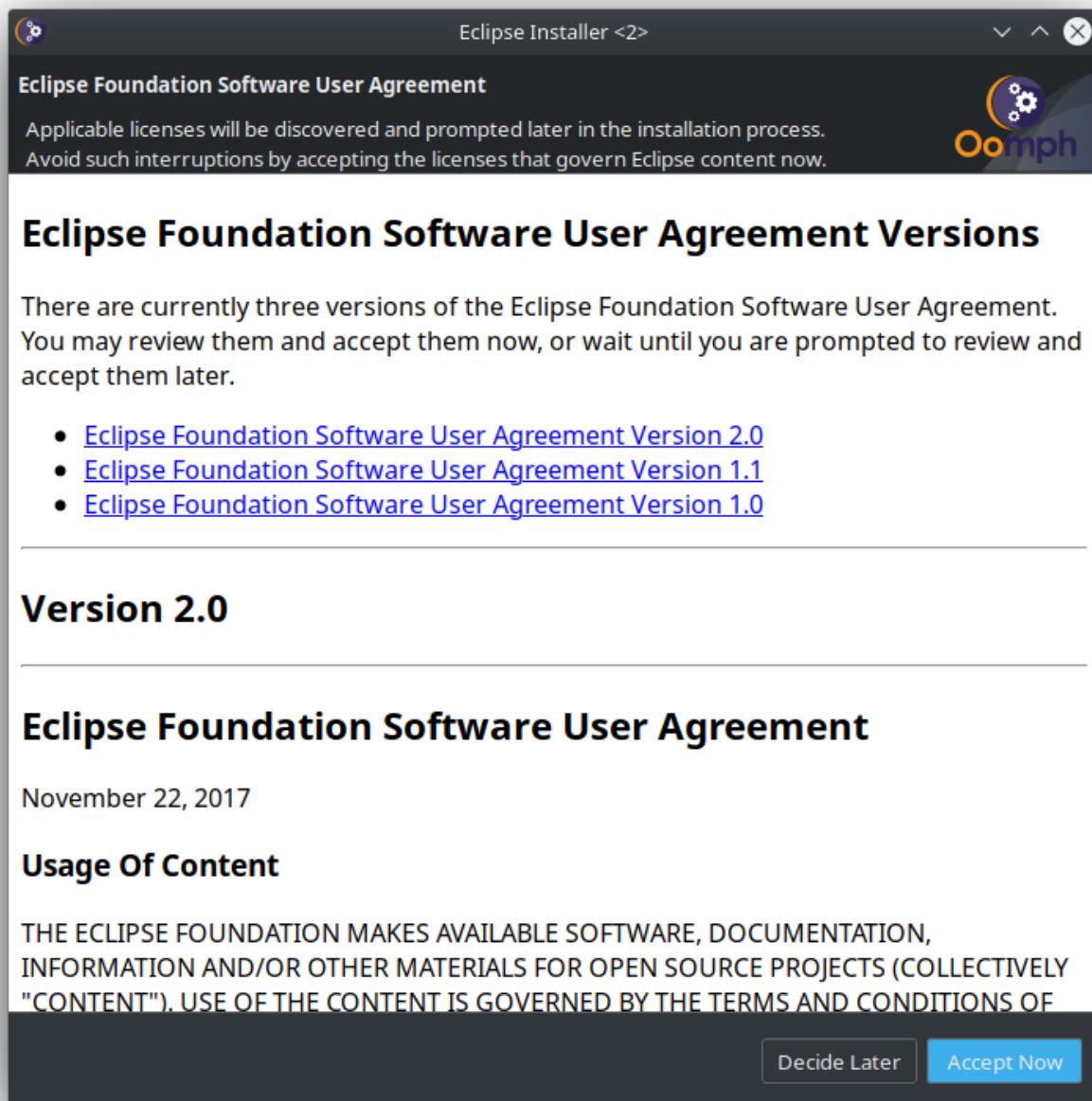
create start menu entry
 create desktop shortcut

INSTALL

BACK

Para seleccionar la carpeta correcta hay que tener en cuenta qué usuarios van a utilizar el entorno. Todos ellos deben tener permisos de lectura y ejecución sobre la carpeta en cuestión. Una vez introducida la carpeta podemos pulsar el botón INSTALAR para iniciar la instalación.

También se nos pedirá que aceptemos las licencias del software a instalar, como muestra la captura de pantalla:



Durante la instalación veremos una pantalla de progreso como la que se muestra a continuación:

The screenshot shows the Eclipse Installer setup window. At the top, there's a "DONATE" button and a gear icon with an exclamation mark. The main title is "eclipseinstaller by Oomph". Below it, there's a section for "Eclipse IDE for Enterprise Java and Web Developers" with a "details" link and a "Java EE IDE" icon.

Configuration fields include:

- Java 11+ VM:** /usr/lib/jvm/java-1.11.0-openjdk-amd64
- Installation Folder:** /root/eclipse/jee-2021-06

Checkboxes for post-installation actions:

- create start menu entry
- create desktop shortcut

A large orange "INSTALL" button is at the bottom, along with a "BACK" link.

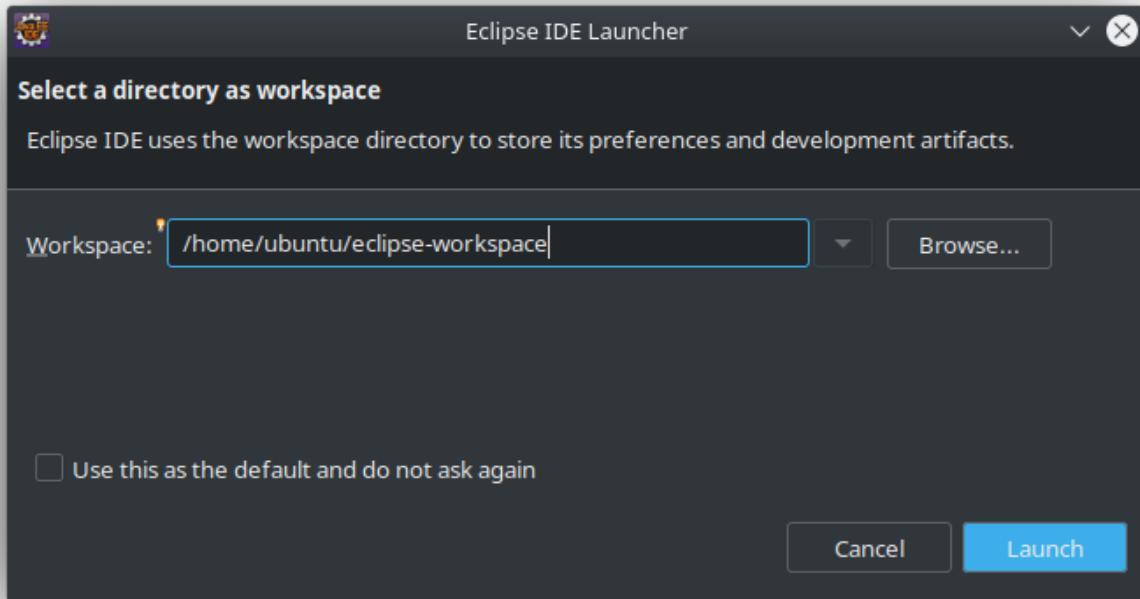
Una vez finalizada la instalación, se nos muestra una pantalla que nos invita a ejecutar directamente el entorno.

The screenshot shows the configuration page for the Eclipse IDE. At the top right are buttons for 'DONATE' and a menu. Below the header, the title 'eclipseinstaller' and 'by Oomph' are displayed. A circular icon for 'Java EE IDE' is shown. The main section is titled 'Eclipse IDE for Enterprise Java and Web Developers' with a 'details' link. It describes tools for Java and Web applications. Configuration fields include 'Java 11+ VM' (set to 'jdk-11.0.11') and 'Installation Folder' (set to '/home/ubuntu/jee-2021-06'). Below these are checkboxes for 'create start menu entry' and 'create desktop shortcut', both of which are checked. A large green 'LAUNCH' button is centered at the bottom, with smaller buttons for 'show readme file' and 'open in system explorer' below it. A 'BACK' button is located at the bottom left.

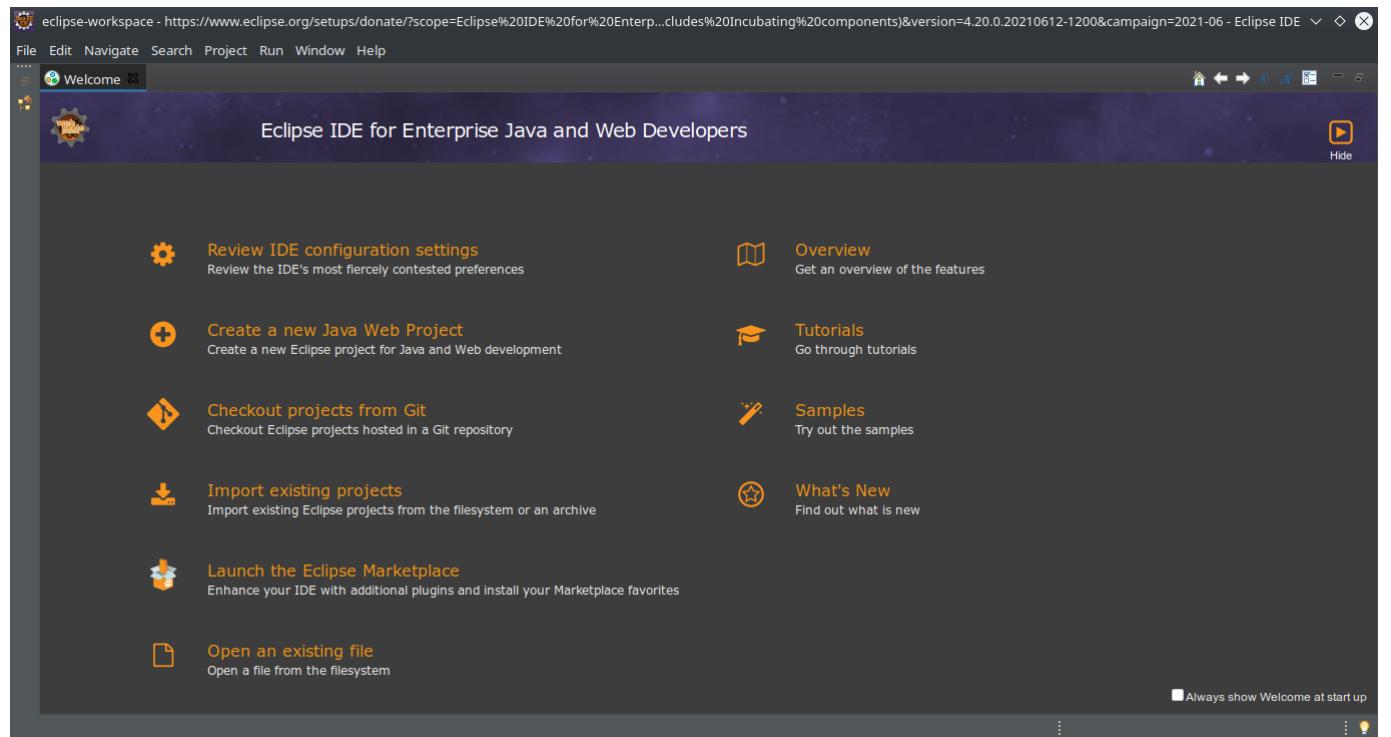
Esta primera vez podremos ejecutar el entorno Eclipse pulsando el botón LAUNCH. El resto de las veces será necesario invocarlo desde los accesos directos o lanzadores, si se han creado o, en caso contrario, invocando directamente el ejecutable. Este se llama eclipse y lo encontrarás en una subcarpeta de la carpeta de instalación también llamada eclipse. La ruta exacta puede variar de una versión a otra. Si en el futuro es necesario desinstalarlo, sólo se debe borrar la carpeta donde ha sido instalado ya que la instalación de Eclipse no aparece en el repositorio de GNU/Linux ni en el panel de control en Windows. Cuando ejecutamos el entorno nos aparecerá una pantalla como la siguiente:



Inmediatamente se nos preguntará en qué carpeta se ubicará el workspace. Podemos pedirle que lo recuerde para el resto de ejecuciones activando la opción “*Usar esto como predeterminado y no volver a preguntar*”.



La primera vez que lo ejecutemos se mostrará la pestaña de bienvenida. Podemos pedirle que no nos la muestre más desactivando la opción "Mostrar siempre la bienvenida al iniciar".



Una vez cerrada esta pestaña, el entorno de trabajo será similar a esto:



Por defecto Eclipse nos ofrece la descarga del instalador más ligero que descargará de Internet los paquetes necesarios para completar la instalación según nuestras elecciones. Si esta instalación nos da problemas, podemos descargar la versión "package" en la que previamente deberemos elegir el paquete de instalación que queramos, ocupará bastante más, pero descargarán todos los paquetes necesarios. Despues solo tendremos que descomprimir el archivo descargado en una carpeta de nuestra elección y ya tendremos eclipse instalado. Tendremos que crear nuestro propio menú de inicio e iconos del escritorio (podéis seguir esta [guía] (<https://www.donovanbrown.com/post/Añadir-Eclipse-al-Launcher-en-Ubuntu-1604>) cambiando la ruta donde habéis descomprimido vuestra versión de eclipse).

5.2.2.2. CONFIGURACIÓN

Versión Java

Por defecto Eclipse intenta utilizar las nuevas características del JDK 16, pero en nuestro caso por ejemplo tenemos la versión 11. Podemos personalizar estas opciones en el apartado Ventana/Preferencias/Java/Compilador y elegir en el campo Nivel de conformidad del compilador la versión correcta, en nuestro caso la 11.

Además, si lo necesitamos, podemos configurar los JDKs que están disponibles, añadirlos o eliminarlos desde la opción Ventana/Preferencias/Java/JRE instalados .

Perspectiva

Eclipse llama a la distribución de los paneles en la ventana Perspectiva, hay unos cuantos predefinidos y podemos configurar los nuestros, a nuestro gusto en la sección Ventana/Perspectiva .

Apariencia

Eclipse nos permite personalizar cualquier aspecto de la apariencia de nuestro entorno, cambiar tanto el tema del IDE como el tamaño de fuente y los colores para el coloreado del código fuente. Todas estas opciones están disponibles en Ventana/Apariencia .

5.2.2.3. MÓDULOS

Las opciones y funcionalidades de Eclipse se pueden ampliar añadiendo módulos desde su "store" de plugins. En Help/Eclipse Marketplace... podemos por ejemplo buscar por texto, o buscar en la pestaña de populares. Eso nos mostrará todos los complementos que contengan la palabra buscada, o los complementos más descargados del marketplace. Podemos instalar, por ejemplo, SonarLint 6.0 que nos ayuda a mantener nuestro código limpio de errores comunes, para ello simplemente tenemos que pulsar el botón INSTALAR que aparece a su lado en el listado, aceptar la licencia de uso y automáticamente nos pedirá que reiniciemos el IDE .

5.2.2.4. USO BÁSICO ("¡HOLA MUNDO!")

Eclipse proporciona información sobre su uso en la sección de `Ayuda`, y podemos aprender a crear nuestro primer proyecto en Java (el típico "¡Hola Mundo!"). Para ello debemos abrir la ventana de `Bienvenido`, que es la que nos aparece cuando abrimos Eclipse por primera vez, o bien podemos abrirla desde `Ayuda/Bienvenido`, desde esta ventana podemos elegir la sección de `Tutoriales`, y dentro de la sección de Desarrollo Java, elegir el primer ítem "Crear una aplicación Hola Mundo", y el propio Eclipse nos irá guiando paso a paso para crear y ejecutar nuestro primer proyecto Java en Eclipse.

5.2.2.5. ACTUALIZACIÓN Y MANTENIMIENTO

En la misma sección `Ayuda` Eclipse nos proporciona las opciones para actualizar el propio Eclipse o los complementos que tengamos instalados `Ayuda/Buscar actualizaciones`.

Podemos personalizar el comportamiento respecto a las actualizaciones en la sección `Ventana/Preferencias/Instalar/Actualizar/Actualizaciones automáticas`.

5.2.3. Netbeans

NetBeans es una herramienta de entorno de desarrollo integrado (IDE) muy potente que se utiliza principalmente para el desarrollo en Java y C/C++. Permite desarrollar fácilmente aplicaciones web, de escritorio y móviles desde su marco modular. Puede agregar soporte para otros lenguajes de programación como PHP, HTML, JavaScript, C, C++, Ajax, JSP, Ruby on Rails, etc. mediante extensiones.

Se ha lanzado NetBeans IDE 12 con soporte para Java JDK 11. También incluye las siguientes características:

- Soporte para PHP 7.0 a 7.3, PHPStan y Twig.
- Incluir módulos en el clúster "webcommon". Es decir, todas las funciones de JavaScript en Apache NetBeans GitHub son parte de Apache NetBeans 10.
- Los módulos de clúster "groovy" están incluidos en Apache NetBeans 10.
- OpenJDK puede detectar automáticamente JTReg desde la configuración de OpenJDK y registrar el JDK expandido como una plataforma Java.
- Soporte para JUnit 5.3.1

5.2.3.1. INSTALACIÓN

Podemos instalar NetBeans de tres maneras:

5.2.3.1.1. Instalar desde binarios

Paso 1: Descargue el archivo NetBeans

Descargue el archivo binario de NetBeans 12 `netbeans-12.4-bin.zip`.

Paso 2: Extraer el archivo

Espere a que finalice la descarga y luego extráigala.

```
1 $ unzip netbeans-12.4-bin.zip
```

Confirme el contenido del archivo de directorio creado:

```
1 $ ls netbeans
2 apisupport enterprise groovy javafx netbeans.css profiler
3 bin ergonomics harness LICENSE NOTICE README.html
4 cpplite etc ide licenses php webcommon
5 DEPENDENCIES extide java nb platform websvccommon
```

Step 3: Move the `netbeans` folder to `/opt`

Ahora movamos la carpeta `netbeans/` a `/opt`

```
1 $ sudo mv netbeans/ /opt/
```

Paso 4: Ruta de configuración

El binario ejecutable de Netbeans se encuentra en `/opt/netbeans/bin/netbeans`. Necesitamos agregar su directorio principal a nuestro `$PATH` para poder iniciar el programa sin especificar la ruta absoluta al archivo binario. Abra su archivo `~/.bashrc` o `~/.zshrc`.

```
1 $ nano ~/.bashrc
```

Añade la siguiente línea al final

```
1 export PATH = "$PATH:/opt/netbeans/bin/"
```

Obtenga el archivo para iniciar Netbeans sin reiniciar el shell.

```
1 $ source ~/.bashrc
```

Paso 5: Crear el iniciador de escritorio NetBeans IDE (opcional)

Cree un nuevo archivo en `/usr/share/applications/netbeans.desktop`.

```
1 $ sudo nano /usr/share/applications/netbeans.desktop
```

Añade los siguientes datos.

```
1 [Desktop Entry]
2 Name=Netbeans IDE
3 Comment=Netbeans IDE
4 Type=Application
5 Encoding=UTF-8
6 Exec=/opt/netbeans/bin/netbeans
7 Icon=/opt/netbeans/nb/netbeans.png
8 Categories=GNOME;Application;Development;
9 Terminal=false
10 StartupNotify=true
```

Para desinstalar NetBeans debemos eliminar la carpeta `netbeans/` que está dentro de la carpeta `/opt/`, podemos utilizar el comando:

```
1 $ sudo rm /opt/netbeans -rf
```

Paso 6: Configurar correctamente el JDK (opcional)

En el fichero `/opt/netbeans/etc/netbeans.conf` debemos especificar correctamente la ruta de nuestro JDK en la variable `netbeans_jdkhome`. En GNU/Linux podemos saber los JDK disponibles con el comando `sudo update-alternatives --config java` que nos mostrará un resultado similar a este:

```
1 Hi ha 3 possibilitats per a l'alternativa java (que proveeix /usr/bin/java).
2
3 Selecció Camí Prioritat Estat
4 -----
5 * 0   /usr/lib/jvm/java-14-openjdk-amd64/bin/java    1411 mode automàtic
6   1   /usr/lib/jvm/java-11-openjdk-amd64/bin/java    1111 mode manual
7   2   /usr/lib/jvm/java-14-openjdk-amd64/bin/java    1411 mode manual
8   3   /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java  1081 mode manual
9
10 Premeu retorn per a mantenir l'opció per defecte[*], o introduiu un número de selecció:
```

En la configuración de netbeans no es necesario especificar el final de la ruta `bin/java`

```
1 netbeans_jdkhome="/usr/lib/jvm/java-11-openjdk-amd64/"
```

5.2.3.1.2. Instalar desde script

Paso 1: Descargue el archivo NetBeans

También puede instalar Netbeans 12.4 en GNU/Linux desde un script proporcionado para descargar `Apache-NetBeans-12.4-bin-linux-x64.sh`.

Paso 2: Ejecutar el script

Debes ejecutar el script de instalación

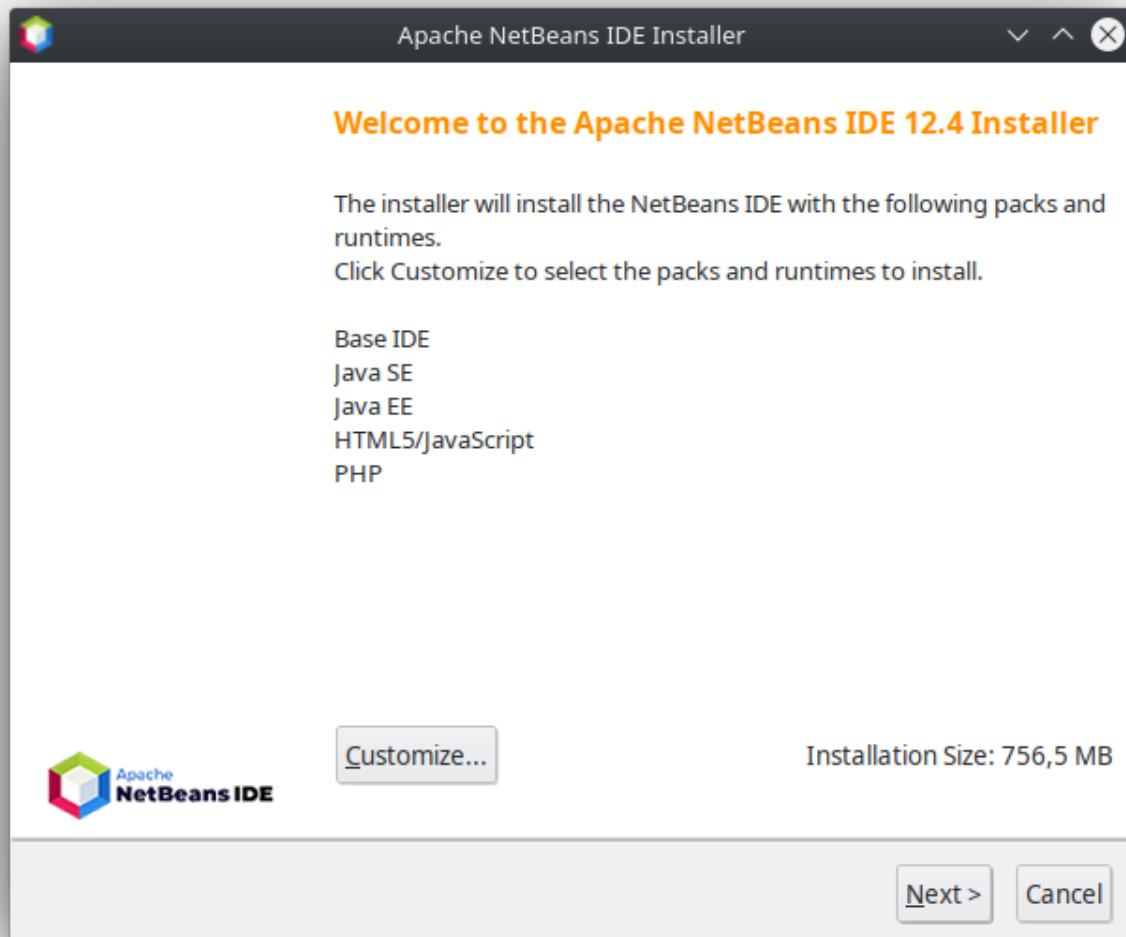
```
1 $ sudo sh ./Apache-NetBeans-12.4-bin-linux-x64.sh
```

Si ejecuta el script como `root` (`sudo`) Netbeans estará disponible para todos los usuarios. Por el contrario, si ejecuta el usuario sin `sudo`, solo estará disponible para su usuario.

Aparecerá una barra de progreso como esta:



Ahora podemos elegir los componentes que queremos instalar con el IDE de Netbeans, lo dejaremos por defecto y pulsaremos el botón siguiente.



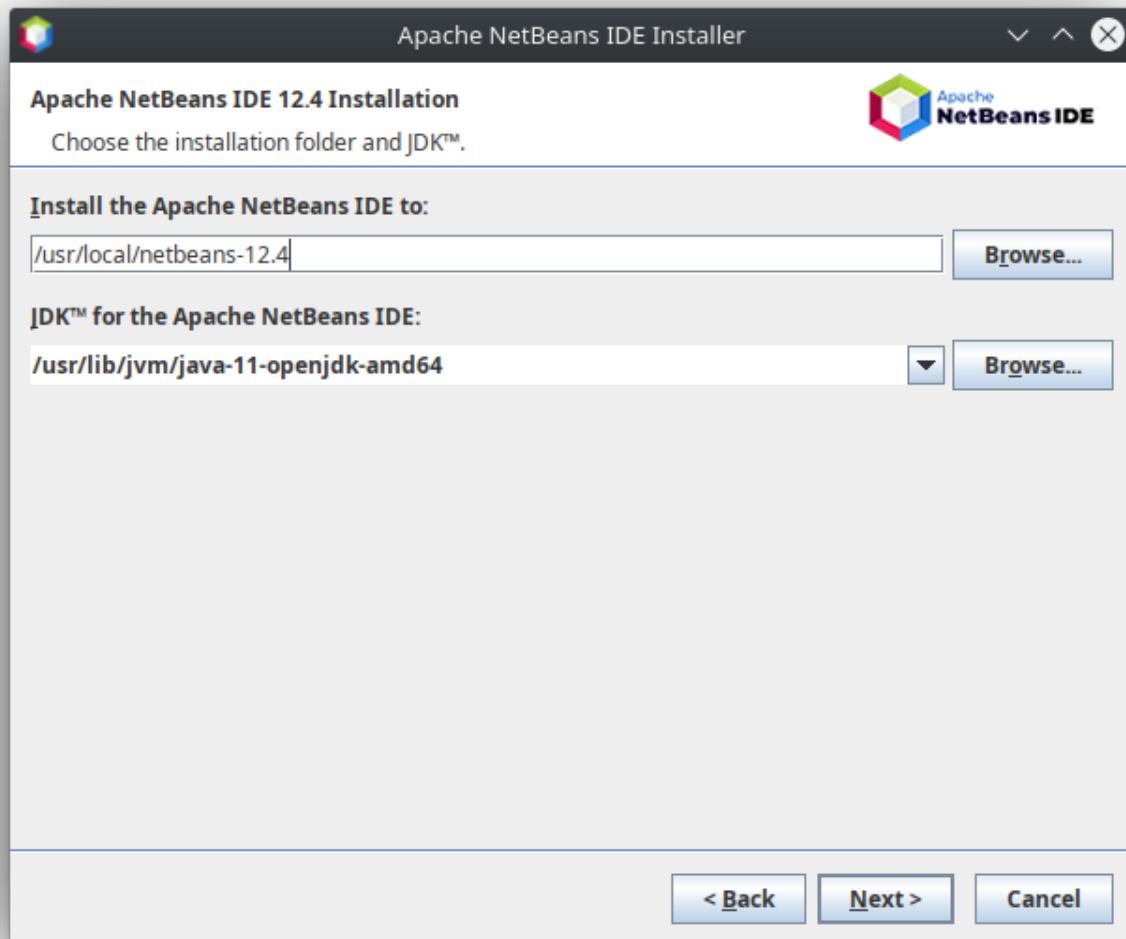
Paso 3: Aceptar la licencia

Luego debemos aceptar el acuerdo de licencia de uso marcando la casilla y presionando el botón siguiente.



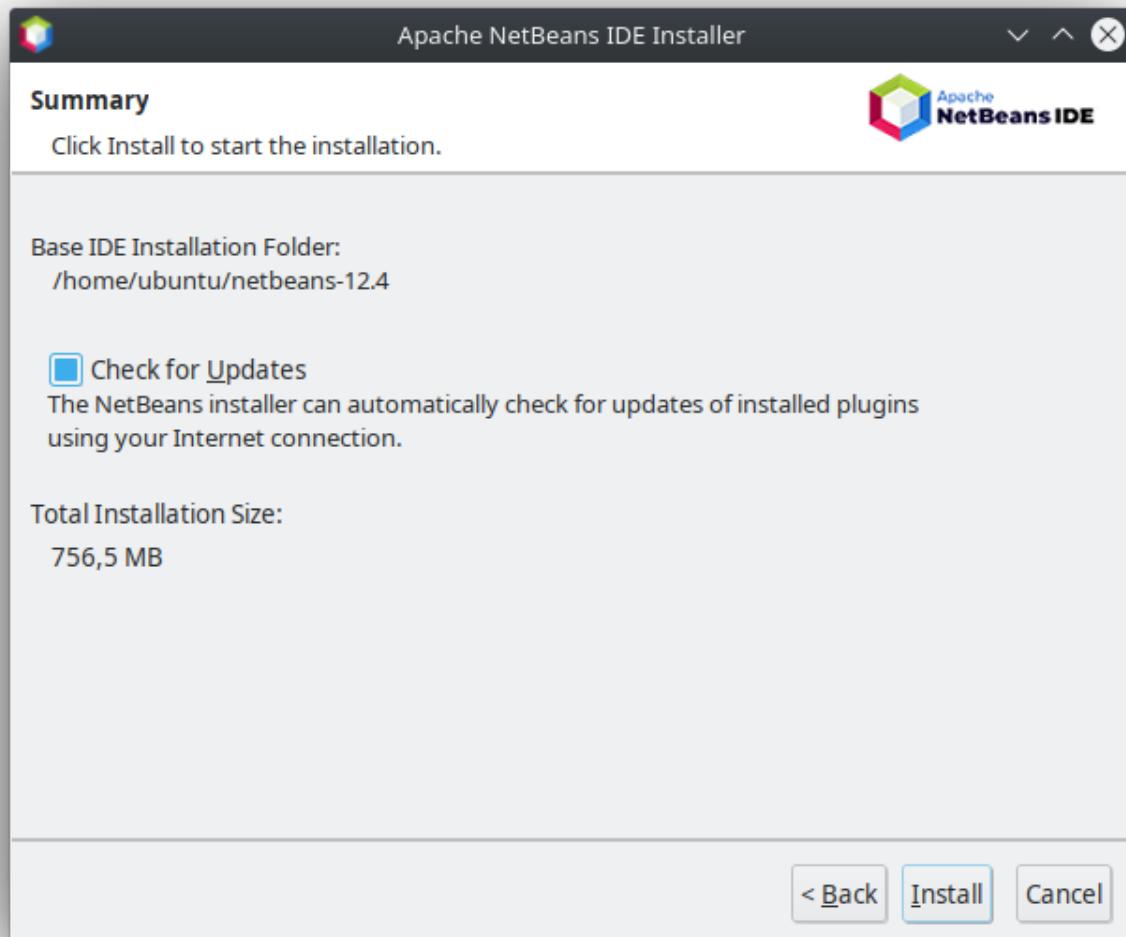
Paso 4: Elija la ruta de instalación y el JDK

Ahora debemos elegir la ruta donde se instalará Netbeans 12.4. Y debemos elegir la ruta donde se encuentra el JDK (por defecto indica `/usr`, pero debemos especificar la ubicación como por ejemplo `/usr/lib/jvm/java-11-openjdk-amd64`).



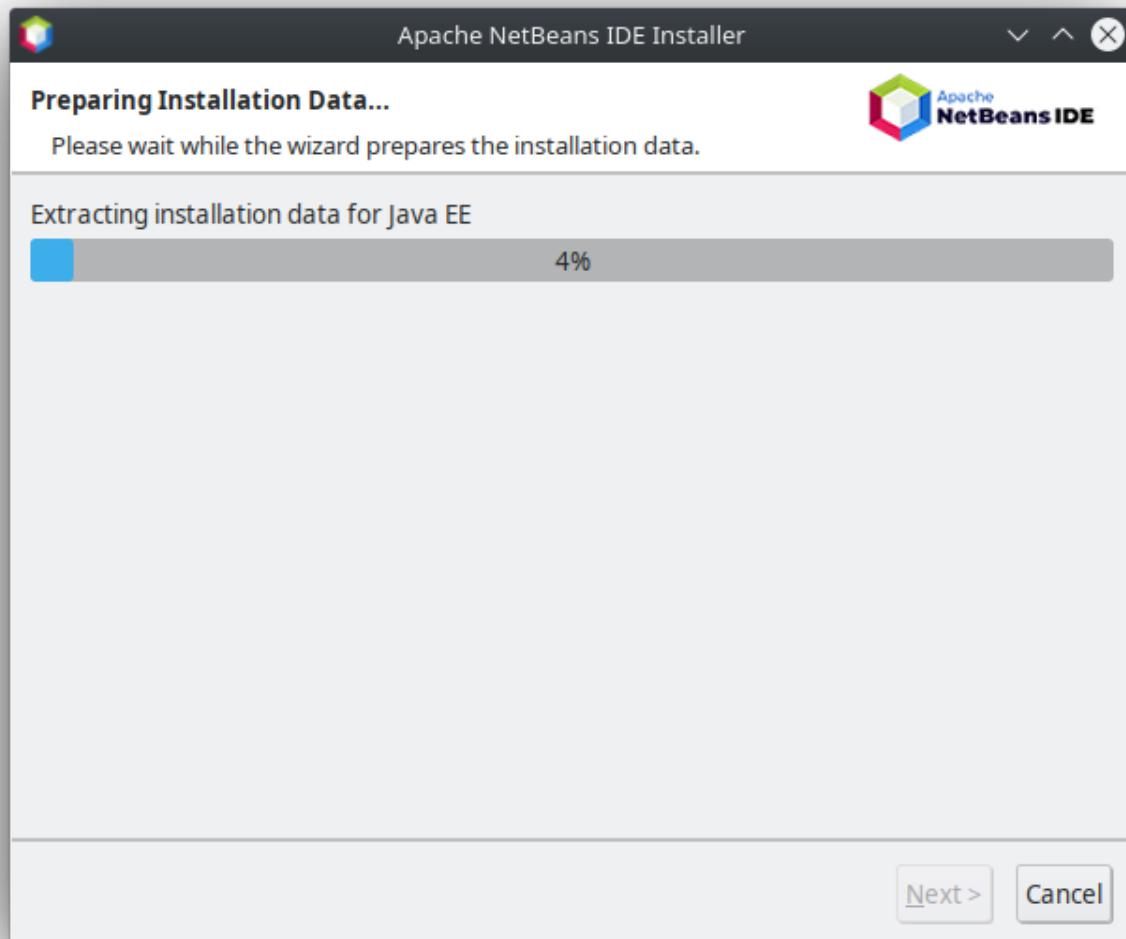
Paso 5: Actualizaciones automáticas

En este punto se muestra un resumen de la instalación, y podemos elegir si queremos que NetBeans busque e instale actualizaciones desde Internet, y pulsar el botón instalar.



Paso 6: Instalación

Aparecerá una barra de progreso.



Paso 7: Paso final

Al terminar, aparecerá una pantalla con las acciones realizadas por el instalador y ya tendremos los launchers creados en el menú de aplicaciones.



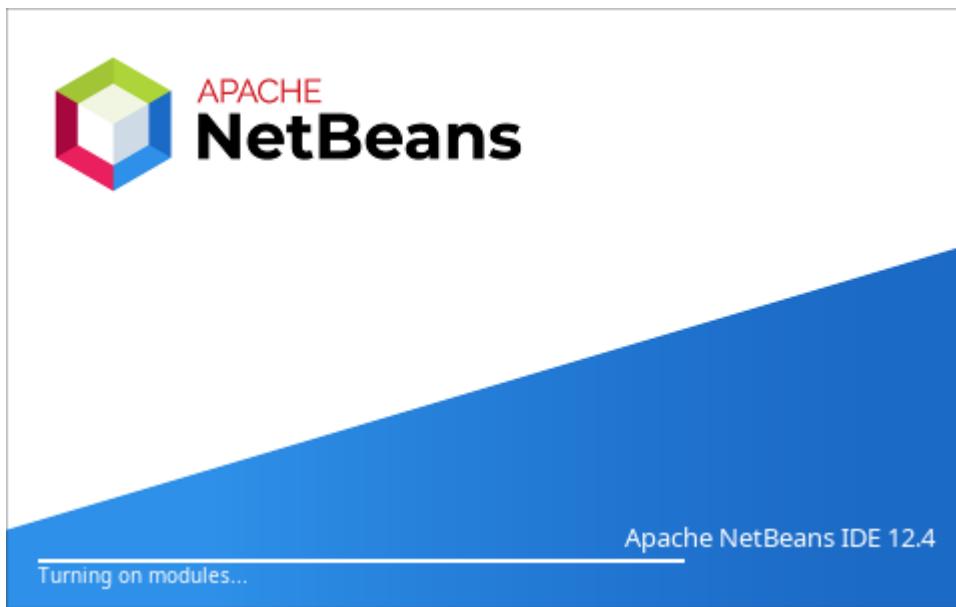
5.2.3.1.3. Instalar mediante snap

Quizás una forma más sencilla de instalar la última versión de Netbeans en nuestro sistema GNU/Linux es a través de `snap`:

```
1 $ sudo snap install netbeans --classic
```

5.2.3.1.4. Primera ejecución

Cuando ejecutamos el entorno nos aparecerá una pantalla como la siguiente:



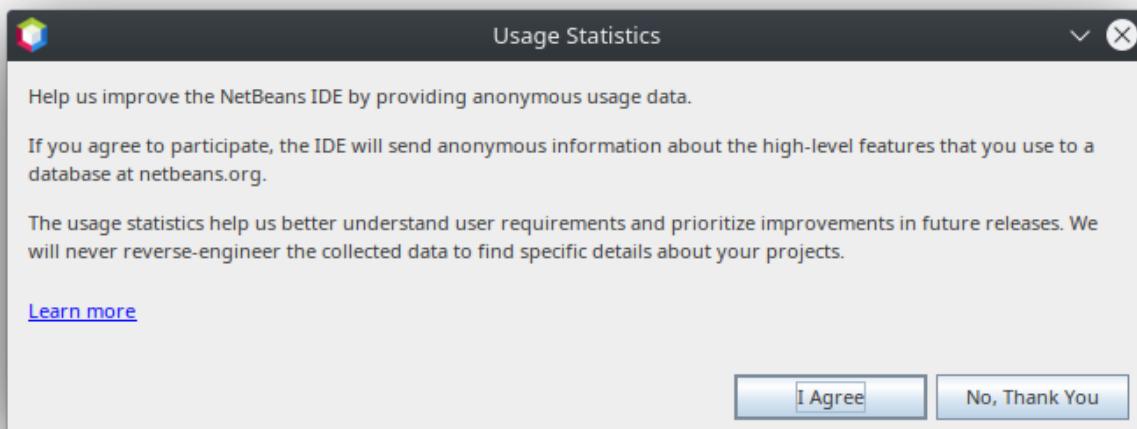
La primera vez que lo ejecutemos se mostrará la pestaña de bienvenida. Podemos pedir que no se nos muestre más desactivando la opción "Mostrar al iniciar".



Una vez cerrada esta pestaña, el entorno de trabajo será similar a esto:



NetBeans puede solicitarnos permiso para utilizar nuestra información a nivel estadístico, elegimos el comportamiento deseado y aceptamos.



Para desinstalar NetBeans en este caso debemos ejecutar el archivo `uninstall.sh` que se encuentra en la carpeta de instalación.

5.2.3.2. CONFIGURACIÓN

Activar módulos

Por defecto Netbeans tiene los módulos desactivados y será la primera vez que los necesitemos cuando pasen a estar activos y disponibles. Por ejemplo, si creamos un nuevo proyecto y elegimos `Java Application` dentro de la categoría `Java with Ant`, veremos en la parte inferior que Netbeans nos avisa de que el módulo necesario no está activo y que debemos pulsar `Next` para

que esté disponible. Lo hacemos, y a continuación nos pedirá que activemos el módulo `nb-javac Impl`, dejamos el check marcado y pulsamos el botón **Activate**, y nos aparecerá el asistente para crear nuestro primer proyecto Java.

Versión Java

Dentro del menú **Herramientas/Plataformas Java** podemos cambiar o ver la ubicación de nuestra instalación JDK.

Perspectiva

En Netbeans las perspectivas no son necesarias, el entorno de Netbeans, aunque es personalizable, se adapta automáticamente a las tareas que estés realizando en cada momento.

Apariencia

Netbeans nos permite personalizar cualquier aspecto de la apariencia de nuestro entorno, cambiar el tema del IDE así como el tamaño de fuente y los colores para el coloreado del código fuente. Todas estas opciones están disponibles en **Herramientas/Opciones**, y dentro de esta ventana elegimos la tercera pestaña **Fuente y Colores** y la penúltima pestaña **Apariencia**.

Configuración de exportación/importación

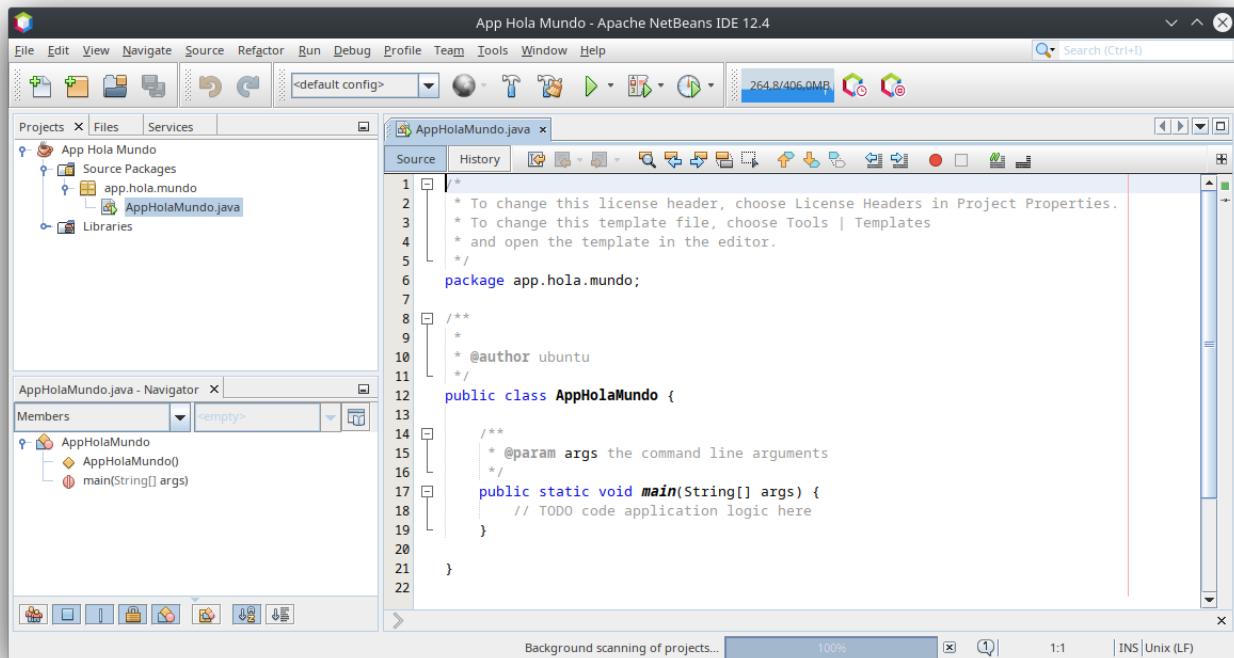
Una opción muy interesante de Netbeans es que nos permite exportar o importar configuraciones y compartirlas con otros compañeros o incluso entre nuestros equipos o diferentes instalaciones. La opción está disponible en **Herramientas/Opciones**, abajo a la izquierda encontramos los botones **Exportar...** e **Importar...**

5.2.3.3. MÓDULOS

Las opciones y funcionalidades de Netbeans se pueden ampliar añadiendo módulos desde su sección de plugins. En **Tools/Plugins** podemos por ejemplo buscar por texto, o buscar en la pestaña de plugins disponibles. Eso nos mostrará todos los plugins que contienen la palabra buscada, o los plugins disponibles. Podemos instalar por ejemplo `sonarlint4netbeans` que nos ayuda a mantener nuestro código limpio de errores comunes, para ello simplemente tenemos que marcar la casilla delante del nombre del plugin, y pulsar el botón **INSTALAR** que aparece más abajo, pulsar siguiente, aceptar la licencia de uso e instalar. Cuando termine la instalación nos pedirá que reiniciemos el **IDE**.

5.2.3.4. USO BÁSICO ("¡HOLA MUNDO!")

Para crear nuestra primera aplicación en Netbeans, debemos crear una aplicación Java, desde el menú **Archivo/Nuevo Proyecto...** debemos elegir **Aplicación Java** dentro de la categoría **Java con Ant**. A continuación debemos especificar el nombre del proyecto, por ejemplo "App Hello World", y nos aseguramos de dejar marcada la opción **Crear clase principal** `app.hello.world.AppHolaWorld` y nos debería aparecer algo como esto:



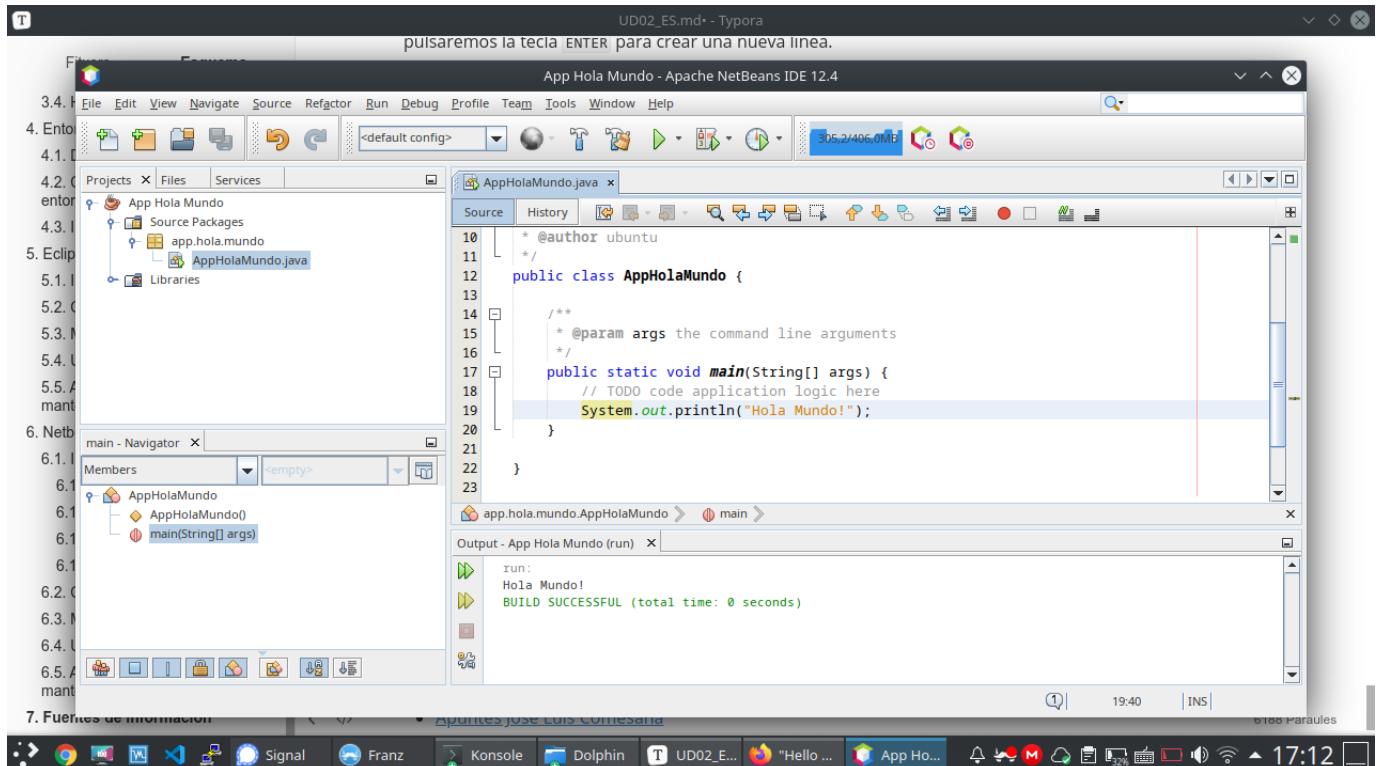
En este punto, sólo nos queda incluir la línea de código necesaria para imprimir el mensaje de texto en pantalla. Para ello, nos dirigiremos al final de la línea `// TODO code application logic here` y pulsaremos la tecla `ENTER` para crear una nueva línea.

Una vez situados en el lugar adecuado utilizaremos una de las funcionalidades más interesantes de Netbeans, que son las plantillas de código. Tecleamos la palabra "sout" y luego pulsamos la tecla `TAB` y Netbeans la sustituirá por el código correcto: `System.out.println ("");`.

Ahora debemos escribir entre las dos comillas dobles el mensaje de texto que debe aparecer en pantalla, y debe quedar así:

```
1 System.out.println("Hola Mundo!");
```

Luego podemos presionar el botón superior con un triángulo verde (`Ejecutar proyecto`) o presionar la tecla `F6` del teclado:



Aparecerá una nueva sección en la ventana (en la parte inferior) llamada `Salida` en la que podremos visualizar el resultado de la ejecución de nuestro primer programa.

5.2.3.5. ACTUALIZACIÓN Y MANTENIMIENTO

En la sección `Ayuda`, Netbeans nos proporciona las opciones para actualizar el propio Netbeans con la opción `Ayuda/Buscar actualizaciones`.

5.2.4. IntelliJ (recomendado)

IntelliJ IDEA es un entorno de desarrollo integrado (IDE) escrito en Java para desarrollar software informático escrito en Java, Kotlin, Groovy y otros lenguajes basados en JVM. Está desarrollado por JetBrains (antes conocido como IntelliJ) y está disponible como una edición comunitaria con licencia Apache 2 y en una edición comercial propietaria. Ambas se pueden utilizar para el desarrollo comercial.

Nuestra institución dispone de licencias para nuestros alumnos mientras tengáis correo electrónico @ieseduardoprimo.es.

5.2.4.1. INSTALACIÓN

Descargue desde <https://www.jetbrains.com/idea/> la versión de la herramienta toolbox correspondiente a su sistema operativo.

Siga las instrucciones para su sistema operativo desde <https://www.jetbrains.com/help/idea/installation-guide.html#toolbox>

Una vez instalada la caja de herramientas, puede elegir instalar todos los productos de JetBrains.

Una vez instalada la Idea (IDE) puedes crear una entrada de escritorio desde la pantalla inicial:



Y en la opción Administrar licencias debes seguir estas instrucciones: https://www.jetbrains.com/help/license_server/Activating_license.html

La dirección del servidor es: <https://iesepm.flx.jetbrains.com/>

5.2.4.2. AJUSTES

Documentos para configurar su IDE: <https://www.jetbrains.com/help/idea/configuring-project-and-ide-settings.html>

5.2.4.3. MÓDULOS

Puedes agregar complementos siguiendo estas instrucciones:

<https://www.jetbrains.com/help/idea/managing-plugins.html>

5.2.4.4. USO BÁSICO ("¡HOLA MUNDO!")

Los documentos te ayudan con tu primer programa en Java: <https://www.jetbrains.com/help/idea/creating-and-running-your-first-java-application.html>

Mucha más información:

- Si vienes de Eclipse: <https://www.jetbrains.com/help/idea/migrating-from-eclipse-to-intellij-idea.html>
- Si estuvieras en NetBeans: <https://www.jetbrains.com/help/idea/netbeans.html>
- Si quieres aprender por tu cuenta: <https://www.jetbrains.com/help/idea/product-educational-tools.html>

5.2.5. Por qué debería elegir IntelliJ en lugar de VsCode para la codificación en Java

5.2.5.1. IDEA INTELLIJ:

Ventajas:

1. **Entorno integrado completo:** IntelliJ IDEA está diseñado específicamente para el desarrollo de Java y ofrece un conjunto completo de herramientas y características optimizadas para esta tarea.
2. **Análisis estático avanzado:** Proporciona un análisis de código en profundidad que detecta errores y problemas potenciales antes de la compilación.
3. **Depuración avanzada:** ofrece un potente conjunto de herramientas de depuración que ayudan a identificar y resolver problemas en el código.
4. **Refactorización guiada:** Proporciona herramientas para reorganizar y optimizar el código de forma segura, promoviendo buenas prácticas de programación.
5. **Compatibilidad con marcos y tecnologías Java:** Integración nativa con muchos marcos y tecnologías utilizados en el desarrollo Java, lo que facilita la creación de aplicaciones completas.
6. **Generación automática de código:** ayuda a los programadores a generar automáticamente fragmentos de código repetitivos, como captadores y definidores.
7. **Integración con herramientas de compilación:** facilita la integración con herramientas de compilación como Maven y Gradle.
8. **Soporte para pruebas unitarias:** Ofrece integración con marcos de prueba como JUnit para el desarrollo basado en pruebas.
9. **Facilidad de configuración:** Proporciona asistentes guiados para configurar de manera eficiente proyectos Java.

Contras:

1. **Mayor consumo de recursos:** Debido a su naturaleza integral y rica en funciones, IntelliJ IDEA puede consumir más recursos del sistema en comparación con IDE más livianos.
2. **Curva de aprendizaje:** Dado que ofrece una amplia gama de funciones, los principiantes pueden tardar un tiempo en familiarizarse con todas las herramientas disponibles.

5.2.5.2. VISUAL STUDIO CODE (VS CODE):

Ventajas:

1. **Ligero y rápido:** VSCode es un editor de código liviano y rápido, lo que lo hace ideal para proyectos más pequeños o para aquellos que prefieren una experiencia más ágil.
2. **Amplia gama de extensiones:** Tiene una amplia comunidad que desarrolla extensiones para diversas tecnologías y lenguajes, incluido Java.
3. **Versatilidad:** Si bien no está diseñado específicamente para Java, se puede personalizar para que funcione con Java a través de extensiones.
4. **Integración de control de versiones:** ofrece integración nativa con sistemas de control de versiones como Git.
5. **Curva de aprendizaje rápida:** Debido a su enfoque más ligero, puede resultar más sencillo para los principiantes comenzar a trabajar con él.

Contras:

1. **Funcionalidad limitada de Java:** Aunque existen extensiones de Java, VSCode no ofrece el mismo conjunto completo de herramientas optimizadas para Java que IntelliJ IDEA.

2. **Análisis menos profundo:** Las capacidades de análisis estático y corrección de código podrían no ser tan avanzadas como las de IntelliJ IDEA.
3. **Depuración limitada:** si bien ofrece depuración, es posible que no sea tan avanzada o completa como la de IntelliJ IDEA.
4. **Configuración manual del proyecto:** La configuración de proyectos Java puede requerir más pasos y configuración manual en comparación con IntelliJ IDEA.

5.2.6. Tarea

Debes entregar un documento `*.pdf` explicando que IDE has elegido para empezar a programar (más adelante lo puedes cambiar si quieres), justificando porqué lo has elegido.

Además envia una captura de pantalla en la que se vea el resultado del comando:

```
1  java --version
```

Y por último capturas de pantalla donde se pueda ver que editas el fichero fuente (`HolaMundo.java`), lo compilas y lo ejecutas dentro del IDE que has elegido (explica los pasos que has seguido)

⌚6 de diciembre de 2025

5.3. Taller UD01_03: Crear cuenta en GitHub

5.3.1. Qué es GitHub

Github es una plataforma en la nube basada en Git que permite a los desarrolladores almacenar, gestionar y colaborar en proyectos de código. Es el portafolio universal de los programadores.

Crear una cuenta es esencial para quien aprende o busca trabajar en programación porque: sirve como tu currículum técnico, donde muestras tus proyectos y evolución; te permite colaborar en proyectos open source para ganar experiencia real; y es una herramienta fundamental para el control de versiones y trabajo en equipo, usada por prácticamente todas las empresas tech.

5.3.2. Crea tu cuenta

Accede a la plataforma GitHub: <https://github.com/>

Pulsa sobre el botón [Sign Up] y sigue las instrucciones para crear tu cuenta.

Una vez creada tu cuenta, entra en tu página principal, por ejemplo la mia es esta: <https://github.com/martinezpenya> (`martinezpenya` es mi usuario de github) y realiza una captura de pantalla.

5.3.3. Solicitar corrección de los apuntes

Ahora, para probar nuestra nueva cuenta y colaborar con algún proyecto, no hay nada mejor que ayudar a mejorar los apuntes del profesor de Programación 😊.

Accedemos a la página de los apuntes en la que hemos detectado el error o queremos sugerir un cambio y en la parte superior derecha debe aparecer el icono:

The screenshot shows a Moodle course page for '1º Programacion (CFGs Desarrollo de Aplicaciones Multiplataforma)'. The left sidebar contains a navigation menu with sections like 'UD00', 'UD01', 'Elementos de un programa informático', 'Ejercicios', and 'Talleres'. Under 'Talleres', there are links for 'T01 NoMachine', 'T02 JDK e IDE', 'T03 GitHub', and 'T04 Markdown'. The main content area displays a section titled 'Taller UD01_T04: Markdown' with a sub-section '1. Introducción a Markdown'. Below this is a large 'MD' logo. To the right, there is a sidebar with a 'Markdown' section containing text about its history and features. At the top right of the page, there is a blue bar with icons for search, GitHub, and other course options, and a red arrow points to the 'Edit this page' button.

Esto nos llevará a crear un Fork del repositorio (este concepto lo aprenderás más adelante en el módulo de Entornos de Desarrollo):

The screenshot shows a GitHub repository page for '1DAMProgramacion / docs / UD01 / UD01_T04_Markdown.md'. The page has a dark theme. At the top, there are navigation links for 'Code', 'Issues', 'Pull requests', 'Actions', 'Projects', 'Security', and 'Insights'. Below the header, it shows the file path and the file name 'UD01_T04_Markdown.md' in the 'main' branch. A large message in the center says 'You need to fork this repository to propose changes.' with a small icon above it. Below the message, a note says 'Sorry, you're not able to edit this repository directly—you need to fork it and propose your changes from there instead.' There are two buttons at the bottom: a green 'Fork this repository' button and a blue 'Learn more about forks' link.

Ahora debemos pulsar el botón **[Fork this repository]**, y a continuación veremos el código de la página en nuestro fork que es `MarkDown` (Puedes aprender más sobre `MarkDown` en el Taller 4):

You're making changes in a project you don't have write access to. Submitting a change will write it to a new branch in your fork [profeDAMCarlet/1DAMProgramación](#), so you can send a pull request.

1DAMProgramacion / docs / UD01 / UD01_T04_Markdown.md in **main**

Commit changes...

```

1 # Taller UD01_T04: Markdown
2
3 ## Introducción a Markdown
4
5 
6
7 **Markdown** nace como herramienta de **conversión de texto plano a HTML**. Fue creada en 2004 por John Gruber, y se distribuye de manera gratuita bajo una \[licencia BSD\](https://es.wikipedia.org/wiki/Licencia\_BSD).
8
9 Markdown es un maravilloso **lenguaje** para escribir documentos de una manera **sencilla de escribir, y que en todo momento mantenga un diseño legible** que contengan elementos como *secciones*, *párrafos*, *listas*, *vínculos* e *imágenes*, *etc*. Pandoc [http://pandoc.org](http://pandoc.org/) ha extendido enormemente la \[sintaxis original de Markdown\](http://daringfireball.net/projects/markdown/) y ha añadido unas pequeñas nuevas características tales como notas al pie de página, citas y tablas. Lo más importante que hace Pandoc es hacer posible la generación de documentos en una amplia variedad de formatos desde Markdown, HTML, LaTeX/PDF, MSWord y Slides.

```

Ahora debemos buscar el texto a modificar y una vez hayamos cambiado algo del documento se activará el botón [Commit changes...]:

You're making changes in a project you don't have write access to. Submitting a change will write it to a new branch in your fork [profeDAMCarlet/1DAMProgramación](#), so you can send a pull request.

1DAMProgramacion / docs / UD01 / UD01_T04_Markdown.md in **main**

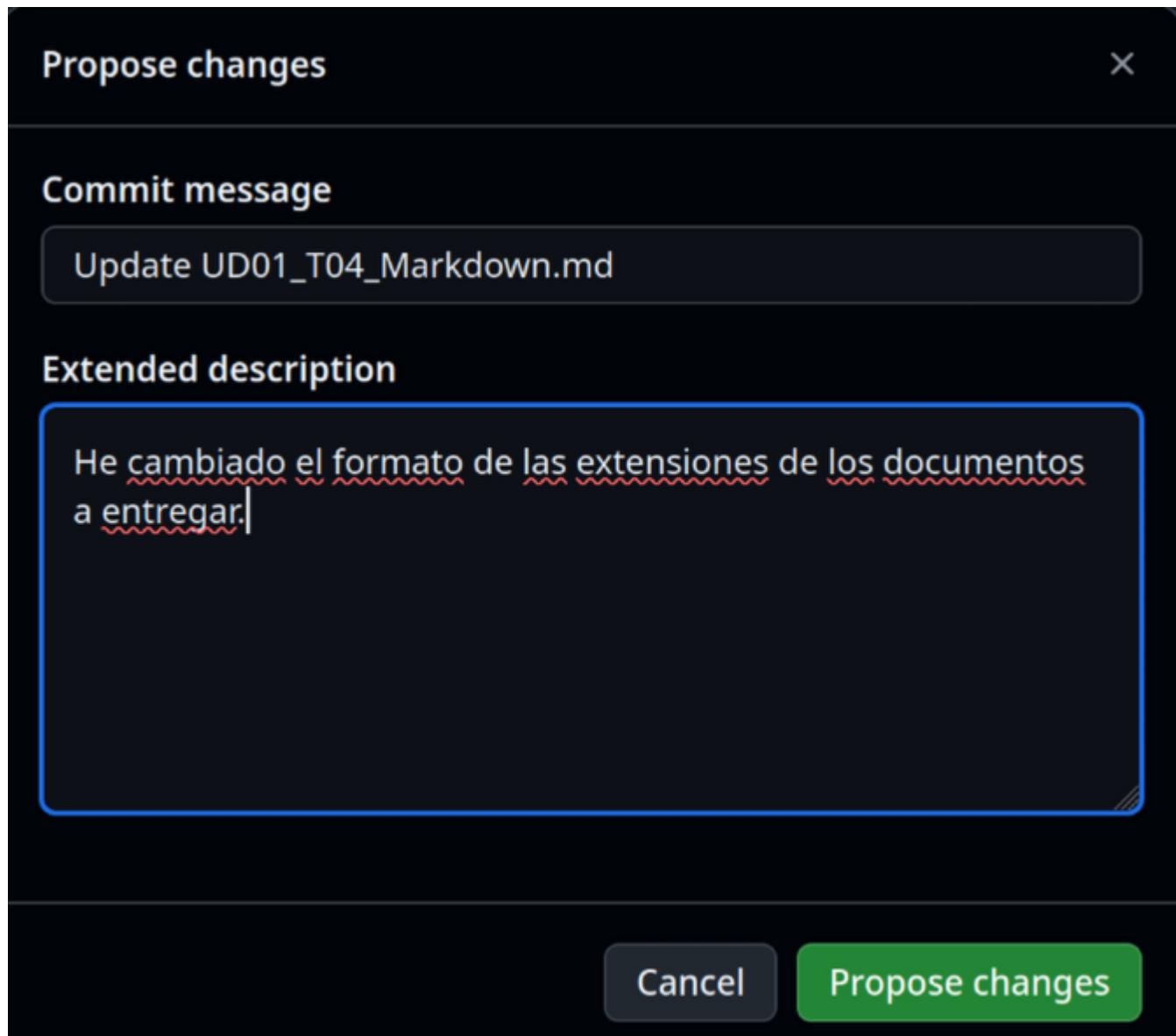
Commit changes...

```

1 # Taller UD01_T04: Markdown
2
3 ## Introducción a Markdown
4
5 
6
7 **Markdown** nace como herramienta de **conversión de texto plano a HTML**. Fue creada en 2004 por John Gruber, y se distribuye de manera gratuita bajo una \[licencia BSD\](https://es.wikipedia.org/wiki/Licencia\_BSD).

```

Ahora debes explicar cual ha sido la modificación que hemos realizado y pulsar el botón [Propose changes]:



Todavía no hemos terminado! ahora hay que comunicar los cambios propuestos en nuestro Fork al propietario del repositorio, para que los visualice y valore si los quiere incluir en la página de documentación. Para ello debemos pulsar el botón [Create pull request]:

The screenshot shows the GitHub interface for comparing changes between two repositories. At the top, it displays the base repository as `martinezpenya/1DAMProgramacion` and the head repository as `profeDAMCarlet/1DAMProgramacion`. The comparison is set to the `main` branch. A red arrow points to the **Create pull request** button.

Below the repository selection, there is a message encouraging users to discuss and review the changes. The comparison summary indicates:

- o 1 commit
- 1 file changed
- 1 contributor

The commit details show a single commit from `profeDAMCarlet` authored 1 minute ago, titled `Update UD01_T04_Markdown.md`. The commit hash is `5f152ae` and it is verified.

The code diff view shows the changes made to the `docs/UD01/UD01_T04_Markdown.md` file. The changes are summarized as:

Showing 1 changed file with 1 addition and 1 deletion.

diff --git a/docs/UD01/UD01_T04_Markdown.md b/docs/UD01/UD01_T04_Markdown.md
--- a/docs/UD01/UD01_T04_Markdown.md 2025-09-06 14:53:40 +0000
+++ b/docs/UD01/UD01_T04_Markdown.md 2025-09-06 14:53:40 +0000
@@ -393,4 +393,4 @@ Como tarea, se propone:

Ahora podemos modificar el mensaje (pero no hace falta), directamente pulsamos sobre el botón [Create pull request]:

The screenshot shows the GitHub interface for creating a pull request. At the top, the repository 'martinezpenya / 1DAMProgramacion' is selected. Below it, the navigation bar includes 'Code' (highlighted), 'Issues', 'Pull requests', 'Actions', 'Projects', 'Security', and 'Insights'. The main section is titled 'Open a pull request' with the sub-instruction 'Create a new pull request by comparing changes across two branches. If you need to, you can also compare across forks. [Learn more about diff comparisons here.](#)' Below this, there are dropdowns for 'base repository' (set to 'martinezpenya/1DAMProgramacion'), 'base' (set to 'main'), 'head repository' (set to 'profeDAMCarlet/1DAMProgramacion'), and 'compare' (set to 'patch-1').

Add a title: A text input field contains the value 'Update UD01_T04_Markdown.md'.

Add a description: A rich text editor window contains the text 'He cambiado el formato de las extensiones de los documentos a entregar.' Below the editor, status messages say 'Markdown is supported' and 'Paste, drop, or click to add files'.

At the bottom right of the editor area, there is a red arrow pointing to the green 'Create pull request' button. This button has a checked checkbox for 'Allow edits by maintainers' and a question mark icon. To the left of the button, a note says 'Remember, contributions to this repository should follow our [GitHub Community Guidelines](#)'.

Ahora si, deberías ver una página similar a la siguiente, de la que también deberás obtener una captura y adjuntarla al .pdf , y además explicar los 4 campos que hay redondeados:

Update UD01_T04_Markdown.md #1

profedAMCarlet wants to merge 1 commit into [martinezpenya:main](#) from [profedAMCarlet:patch-1](#)

Conversation 0 Commits 1 Checks 0 Files changed 1 +1 -1 0 0

profedAMCarlet commented now

He cambiado el formato de las extensiones de los documentos a entregar.

No conflicts with base branch

Changes can be cleanly merged.

Add a comment

Write Preview

Add your comment here...

Markdown is supported Paste, drop, or click to add files

[Close pull request](#) [Comment](#)

Remember, contributions to this repository should follow our [GitHub Community Guidelines](#).

ProTip! Add `.patch` or `.diff` to the end of URLs for Git's plaintext views.

Reviewers
No reviews
Still in progress? [Convert to draft](#)

Assignees
No one assigned

Labels
None yet

Projects
None yet

Milestone
No milestone

Development
Successfully merging this pull request may close these issues.
None yet

Notifications [Unsubscribe](#)
You're receiving notifications because you authored the thread.

1 participant

Allow edits by maintainers [?](#)

Como resumen:

1. Hemos creado un fork de un repositorio.
2. Hemos modificado un archivo en nuestro fork.
3. Hemos comparado nuestro fork con el original y hemos creado un pull request con las diferencias.

Ahora pueden pasar dos cosas, que el propietario del repositorio original acepte nuestros cambios, y por tanto pasaremos a ser colaboradores del repositorio original.

O bien, que el cambio no sea aceptado.

En cualquiera de los dos casos, si adjuntas las capturas y explicas los campos la actividad estará correcta.

En este caso concreto se ha aceptado la modificación:

Update UD01_T04_Markdown.md #1

Merged martinezpenya merged 1 commit into `martinezpenya:main` from `profeDAMCarlet:patch-1` 1 minute ago

Conversation 1 Commits 1 Checks 0 Files changed 1

profeDAMCarlet commented 10 minutes ago

He cambiado el formato de las extensiones de los documentos a entregar.

martinezpenya commented 1 minute ago

Perfecto, gracias!

martinezpenya closed this 1 minute ago

martinezpenya merged commit `8052475` into `martinezpenya:main` 1 minute ago

5.3.4. Tarea

- Crea un documento `.pdf` donde debes adjuntar la captura de tu perfil de github.
- Añade una **captura** de pantalla donde se vea que has solicitado el **pull request** y que estás esperando a que se integre en el repositorio original.
- Además, **explica** que significan cada uno de los **4 apartados** señalados en la captura.

Adjunta el documento `.pdf` con las capturas y las explicaciones a la tarea de AULES.

29 de septiembre de 2025

5.4. Taller UD01_T03: Markdown

5.4.1. Introducción a Markdown



Markdown nace como herramienta de **conversión de texto plano a HTML**. Fue creada en 2004 por John Gruber, y se distribuye de manera gratuita bajo una [licencia BSD](#).

Markdown es un maravilloso **lenguaje** para escribir documentos de una manera **sencilla de escribir, y que en todo momento mantenga un diseño legible** que contengan elementos como *secciones, párrafos, listas, vínculos e imágenes, etc.* Pandoc <http://pandoc.org> ha extendido enormemente la [sintaxis original de Markdown](#) y ha añadido unas pequeñas nuevas características tales como notas al pie de página, citas y tablas. Lo más importante que hace Pandoc es hacer posible la generación de documentos en una amplia variedad de formatos desde Markdown, HTML, LaTeX/PDF, MSWord y Slides.

Este método te permitirá añadir formatos tales como **negritas, cursivas o enlaces**, utilizando texto plano, lo que permitirá hacer de tu escritura algo más simple y eficiente al evitar distracciones.

Con Markdown **no vas a reemplazar todo**, sino cubrir las funcionalidades más comunes que se requieren para escribir un documento relativamente complicado.

5.4.2. Para qué sirve Markdown

Markdown será perfecto para ti sobre todo **si publicas de manera constante en Internet**, donde el lenguaje HTML está más que presente: WordPress, Squarespace, Jekyll...

Pero no estoy hablando solo de [blogs](#) o páginas web. **Servicios** como Trello o **foros** como Stackoverflow también soportan este lenguaje, y con el paso del tiempo encontrarás aún más lugares que lo utilicen.

Además, Markdown está cada vez más extendido en el **mundo “offline”**. Nada te impedirá utilizar este lenguaje para **tomar notas y apuntes** de tus clases o reuniones en una determinada **aplicación** (incluso podrías **escribir un libro con él**, ya que puedes exportar fácilmente el resultado final a un formato ePub).

Gracias a la simplicidad de su sintaxis podrás utilizarlo siempre que necesites escribir y dar formato rápidamente, sobre todo si quieras hacerlo desde dispositivos móviles.

5.4.3. Por qué utilizar Markdown

5.4.3.1. VENTAJAS

- **Markdown para todo.** Para crear apuntes, documentos, notas, sitios web, libros, documentación técnica, etc. de forma off-line.
- **Markdown transportable.** Este tipo de formato siempre será **compatible con todas las plataformas** que utilices, así que utilizar Markdown es una manera de mantener todo tu contenido siempre accesible desde cualquier dispositivo (smartphones, ordenadores de escritorio, tablets...), ya que en cualquiera de ellas siempre encontrarás **las aplicaciones adecuadas** para leer y editar este tipo de contenido.
- Ideal para escribir un libro, pues permite la exportación fácil en ePub, PDF...

Si en el futuro Microsoft Word desapareciese perderías acceso a todo el contenido que has creado durante años utilizando dicho procesador. Así que lo más inteligente para evitar eso es **generar tu contenido de la manera más sencilla posible**: utilizando texto plano.

5.4.3.2. DESVENTAJAS

- No tiene muchas funcionalidades (esto es lo que lo hace muy compatible).
- Al no tener todas las opciones de un procesador de textos a veces tendrás que combinar Markdown con HTML para lograr ciertos formatos.

5.4.4. Editores para Markdown

5.4.4.1. OFF-LINE

- **Typora**
- MarkdownPad
- HarooPad
- Markdown Monster
- ...

5.4.4.2. ONLINE

- Dillinger
- GitHub
- ...

5.4.5. Párrafos y saltos de línea

Si queremos generar un nuevo párrafo en Markdown simplemente separa el texto mediante una línea en blanco (**pulsando dos veces intro**).

Al igual que sucede con HTML, **Markdown no soporta dobles líneas en blanco**, así que si intentas generarlas estas se convertirán en una sola al procesarse.

Para realizar un salto de línea y empezar **una frase en una línea siguiente dentro del mismo párrafo**, tendrás que pulsar **dos veces la barra espaciadora antes de pulsar una vez intro**.

Por ejemplo si quisieses escribir un poema quedaría tal que así:

«*La tierra estaba seca, No había ríos ni fuentes. Y brotó de tus ojos.*

Donde cada verso tiene **dos espacios en blanco al final**.

5.4.6. Encabezados

Las **# almohadillas** son uno de los métodos utilizados en Markdown para crear encabezados. Debes usarlos añadiendo **uno por cada nivel**.

Es decir,

```
1 # Encabezado 1
2 ## Encabezado 2
3 ### Encabezado 3
4 #### Encabezado 4
5 ##### Encabezado 5
6 ##### Encabezado 6
```

Se corresponde con:

1. Encapçalament 1

1.1. Encapçalament 2

1.1.1. Encapçalament 3

1.1.1.1. Encapçalament 4

1.1.1.1.0.1. Encapçalament 5

1.1.1.1.0.1. Encapçalament 6

También puedes cerrar los encabezados con el mismo número de almohadillas, por ejemplo escribiendo `### Encabezado 3 ###`. Pero la única finalidad de esto es un **motivo estético**.

5.4.7. Texto básico

Un párrafo no requiere sintaxis especial.

Para aplicar **negrita** al texto, se escribe entre dos asteriscos.

Para aplicar *cursiva* al texto, se escribe entre un solo asterisco.

Para tachar el texto, se escribirá dos virgullillas antes y dos después de éste.

```
1 Este texto es en **negrita**.
2 Este texto es en *itálica*.
3 Este texto está ~~tachado~~.
4 Este texto es en ambos ***negrita e itálica***.
```

Se corresponde a:

Este texto es en ****negrita****.

Este texto es en *itálica*.

Este texto está ~~tachado~~.

Este texto es en ambos ***negrita e itálica***.

En Markdown no podemos subrayar el texto. Sin embargo, podremos añadir la etiqueta de html underline \u.

```
1 Este texto está <u>subrayado</u>
```

Este texto está subrayado

Para **ignorar los caracteres** de formato de Markdown, ponga \ antes del carácter:

5.4.8. Citas

Las citas se generar utilizando el carácter *mayor que* > al comienzo del bloque de texto.

```
1 > No hay que ir para atrás ni para darse impulso. — Lao Tsé.
```

No hay que ir para atrás ni para darse impulso. — Lao Tsé.

Si la cita en cuestión se compone de **varios párrafos**, deberás añadir el mismo símbolo > al comienzo de cada uno de ellos.

5.4.9. Listas

5.4.9.1. LISTAS ORDENADAS

Para crear **listas numeradas**, empieza una línea con `1.` or `1)`.

No debes mezclar los formatos dentro de la misma lista. No es necesario especificar los números. GitHub lo hace por tí.

```
1 1. ítem 1 de la lista.
2 1. Siguiente ítem de la lista.
3 1. Siguiente ítem, el tercero, de la lista.
```

Se corresponde con:

- 1. Ítem 1 de la lista.
- 2. Siguiente ítem de la lista.
- 3. Siguiente ítem, el tercero, de la lista.

5.4.9.2. LISTAS NO ORDENADAS

Para crear listas no numeradas, o de viñetas, empieza una línea con `*` , `-` o `+` , pero no mezcles los formatos dentro de la misma lista. (No mezclar formatos de viñetas, como `*` y `+` por ejemplo, dentro del mismo documento).

```
1 * ítem 1 de la lista.
2 * Siguiente ítem de la lista.
3 * Siguiente ítem, el tercero, de la lista.
```

Se corresponde con:

- Ítem 1 de la lista.
- Siguiente ítem de la lista.
- Siguiente ítem, el tercero, de la lista.

También podremos combinar ambos tipos de listas. Como por ejemplo:

1. element de llista 2 - element de llista 2.2
 - element de llista 2.2.1
 - element de llista 2.2.2

5.4.9.3. LISTAS DE TAREAS

Para crear listas de tareas basta con que empiece la línea con `- []` , si queremos que no esté el check marcado, y `- [x]` , si queremos que esté el check marcado.

```
1 - [x] regar plantas.
2 - [ ] realizar ejercicios de programación.
```

Se corresponde con:

- regar plantas.
- realizar ejercicios de programación.

5.4.10. Tablas

Las tablas no forman parte de la especificación principal de Markdown, pero Adobe, en cierta forma, las admite.

Para generar una tabla utiliza la barra vertical `|` para generar filas y columnas.

Si insertamos guiones `---` dentro de una celda crearemos el encabezado de la tabla.

```

1 | encabezado1 | encabezado2 | encabezado3 |
2 |---|---|---|
3 | celda 1.1 | celda 1.2 | celda 1.3 |
4 | celda 2.1 | celda 2.2 | celda 2.3 |

```

Quedaría:

encabezado1	encabezado2	encabezado3
celda 1.1	celda 1.2	celda 1.3
celda 2.1	celda 2.2	celda 2.3

Si queremos una **celda con más de una línea** de texto podemos insertar \n (o **Shift+Intro**) al final de ésta.

5.4.11. Enlaces

Para generar enlace en Markdown se debe poner un código con dos partes:

- [texto del enlace], que es el texto que se va a mostrar,
- Y después (nombrefichero.md), que es la URL o el nombre de archivo al que se va a vincular.

```

1 [link text](file-name.md)

```

Un ejemplo:

[enlace a web del centro](https://iesmre.com)

La visualización del ejemplo anterior:

[enlace a web del centro](https://iesmre.com)

5.4.12. Imágenes

Para insertar una imagen se debe poner un código con dos partes:

- ! [texto alternativo], que es el texto que se va a mostrar si la imagen no pudiera visualizarse,
- Seguido de (nombrefichero.extension), que es el archivo imagen (con su dirección).

```

1 [texto alternativo](file-name.md)

```

Un ejemplo:

[logo markdown](assets/mardown_logo.png)

La visualización de la imagen anterior:



5.4.13. Código de bloque

Uno de los puntos más útiles de Markdown a la hora de crear un documento con texto específico de informática es que admite la colocación de bloques de código tanto en línea como en un bloque "delimitado" independiente entre frases.

Para ello utilizaremos:

- Dos comillas invertidas `` si queremos escribir código dentro de la misma línea de texto del párrafo.
- Si queremos crear un bloque de código multilínea, con un lenguaje específico, pondremos ``` seguido del nombre del lenguaje del bloque .

Unos ejemplos:

- En la misma línea:

...estamos escribiendo un párrafo ``insertar el bloque y seguimos escribiendo...

- Un bloque de código:

```javascript y escribimos el código.

```
1 function holamundo(){
2 console.log ("hola mundo web");
3 }
```

#### 5.4.14. Línea horizontal

Para crear una línea horizontal, de separación de contenido por ejemplo, se añaden tres guiones: ---

Visualización:

---

#### 5.4.15. Insertar emojis

Para insertar emojis basta con utilizar `:` seguido del nombre del emoji y cerrar con otro `:`.

Podemos observar, que en algunos editores markdown, al escribir, por ejemplo, `:a` nos muestra todos los emojis con la inicial **a**.

Por ejemplo: `: star :`

Visualización: 

#### 5.4.16. Crear diagrama de flujo

Cuando queremos crear documentos con elementos gráficos como diagramas de flujo, debemos generar una especie de *código* para construirlos.

- Por eso, comenzaremos introduciendo la línea de inicio: ````flow`
- Es conveniente asignar un nombre (por ejemplo: st, op, cond, e...) a cada elemento que conforma el diagrama; así, después podremos unir todos estos.
- Forma de inicio: `st=>start: Nombre`
- Forma de fin: `e=>end: Nombre`
- Rectángulo: `op=>operation: texto de nombre`
- Condición: `cond=>condition: texto de la condición (Sí o No?)`
- Subrutina: `sub1=>subroutine: nombre subtarea`
- EntradaSalida: `io1=>inputoutput: nombre elemento entrada/salida`
- Líneas: `st->op->cond`
- Caminos de condiciones: `cond(yes)**->**e` y `cond(no)->op`
- Línea de cierre: `````

Ejemplo:

```

1 ```flow
2 st=>start: Usuario
3 e=>end: Acceso
4 op=>operation: Operación de usuario
5 cond=>condition: Sí o No?
6 st->op->cond
7 cond(yes)->e
8 cond(no)->op
9 ```


```

Visualización:



Intenta realizar un diagrama para "programar" un almuerzo. En él, deberás dar los **buenos días**, indicar que **es hora del descanso**, y preguntar si **alguién quiere almorzar**. Si no hay nadie que quiera almorzar contigo, debes **ir a otro grupo de amigos** y volver a indicar que **es hora del descanso**. Si alguien sí quiere almorzar **escribe en la pizarra que os vais a almorzar y sal al patio**.

#### 5.4.17. Crear secuencias

En la secuenciación podemos observar que es bastante parecido a la creación de diagramas; pero la primera línea (crear un bloque de código) no será **flow** sino **sequence**.

```

1 ``sequence
2 Ana->Mundo: Hola Mundo
3 Note right of Mundo: Mundo está pensando\nla respuesta
4 Mundo-->Ana: Cómo estás?
5 Ana->>Mundo: Estoy bien gracias!
6 ```

```

Visualización:



#### 5.4.18. Crear índice

Para crear el índice a partir de los encabezados creados debemos insertar `[TOC]`.

#### 5.4.19. Tarea

Como tarea, se propone:

- Crear un documento markdown en tu editor markdown favorito (por ejemplo Typora o VSCode) que documente información acerca de tí mismo.
- En dicho documento crear título, índice.
- Añadir 4 encabezados principales (y otros encabezados secundarios dentro de éstos) en el que hables por ejemplo de: *Tus datos, Currículum, Aficiones y Otros datos de interés*. No hace falta que indiques información personal relevante. (O te la puedes inventar)
- Se valorará la inclusión de distintos elementos como: negrita-cursiva-subrayado, listas ordenadas-desordenadas-tareas, enlaces, imágenes, citas, código, etc.
- Si te atreves con ello, crea un diagrama de flujo en el que indiques los pasos que realizas un sábado por la mañana.
- Exporta el documento a pdf.

**Subir a la plataforma **AULES** un documento Markdown (.md) y otro documento PDF (.pdf) que sea la exportación del primero.**

⌚23 de septiembre de 2025

## 3. UD02

### 6. 3.1 Utilización de Objetos y Clases



#### 6.1. Introducción a la POO

**Orientado a objetos** hace referencia a una forma diferente de acometer la tarea del desarrollo de software, frente a otros modelos como el de la programación imperativa, la programación funcional o la programación lógica. Supone una reconsideración de los métodos de programación, de la forma de estructurar la información y, ante todo, de la forma de pensar en la resolución de problemas.

La **programación orientada a objetos (POO)** es un modelo para la elaboración de programas que ha impuesto en los últimos años. Este auge se debe, en parte, a que esta forma de programar está fuertemente basada en la representación de la realidad; pero también a que refuerza el uso de buenos criterios aplicables al desarrollo de programas.

#### Acción

La orientación a objetos no es un tipo de lenguaje de programación. Es una metodología de trabajo para crear programas.

En POO, un programa es una colección de objetos que se relacionan entre sí de distintas formas.

#### 6.2. Características de la POO

Cuando hablamos de Programación Orientada a Objetos, existen una serie de características que se deben cumplir. Cualquier lenguaje de programación orientado a objetos las debe contemplar. Las características más importantes del paradigma de la programación orientada a objetos son:

- **Abstracción.** Es el proceso por el cual definimos las características más importantes de un objeto, sin preocuparnos de cómo se escribirán en el código del programa, simplemente lo definimos de forma general. En la Programación Orientada a Objetos la herramienta más importante para soportar la abstracción es la clase. Básicamente, una clase es un tipo de dato que agrupa las características comunes de un conjunto de objetos. Poder ver los objetos del mundo real que deseamos trasladar a nuestros

programas, en términos abstractos, resulta de gran utilidad para un buen diseño del software, ya que nos ayuda a comprender mejor el problema y a tener una visión global de todo el conjunto. Por ejemplo, si pensamos en una clase Vehículo que agrupa las características comunes de todos ellos, a partir de dicha clase podríamos crear objetos como Coche y Camión. Entonces se dice que Vehículo es una abstracción de Coche y de Camión.

- **Modularidad.** Una vez que hemos representado el escenario del problema en nuestra aplicación, tenemos como resultado un conjunto de objetos software a utilizar. Este conjunto de objetos se crean a partir de una o varias clases. Cada clase se encuentra en un archivo diferente, por lo que la modularidad nos permite modificar las características de la clase que define un objeto, sin que esto afecte al resto de clases de la aplicación.

- **Encapsulación.** También llamada "ocultamiento de la información". La encapsulación o encapsulamiento es el mecanismo básico para ocultar la información de las partes internas de un objeto a los demás objetos de la aplicación. Con la encapsulación un objeto puede ocultar la información que contiene al mundo exterior, o bien restringir el acceso a la misma para evitar ser manipulado de forma inadecuada. Por ejemplo, pensemos en un programa con dos objetos, un objeto Persona y otro Coche. Persona se comunica con el objeto Coche para llegar a su destino, utilizando para ello las acciones que Coche tenga definidas como por ejemplo conducir. Es decir, Persona utiliza Coche pero no sabe cómo funciona internamente, sólo sabe utilizar sus métodos o acciones.

- **Jerarquía.** Mediante esta propiedad podemos definir relaciones de jerarquías entre clases y objetos. Las dos jerarquías más importantes son la jerarquía "es un" llamada generalización o especialización y la jerarquía "es parte de", llamada agregación. Conviene detallar algunos aspectos:

- La generalización o especialización, también conocida como herencia, permite crear una clase nueva en términos de una clase ya existente (herencia simple) o de varias clases ya existentes (herencia múltiple). Por ejemplo, podemos crear la clase CochedeCarreras a partir de la clase Coche, y así sólo tendremos que definir las nuevas características que tenga.
- La agregación, también conocida como inclusión, permite agrupar objetos relacionados entre sí dentro de una clase. Así, un Coche está formado por Motor, Ruedas, Frenos y Ventanas. Se dice que Coche es una agregación y Motor, Ruedas, Frenos y Ventanas son agregados de Coche.
- **Polimorfismo.** Esta propiedad indica la capacidad de que varias clases creadas a partir de una antecesora realicen una misma acción de forma diferente. Por ejemplo, pensemos en la clase Animal y la acción de expresarse. Nos encontramos que cada tipo de Animal puede hacerlo de manera distinta, los Perros ladran, los Gatos maullan, las Personas hablamos, etc. Dicho de otra manera, el polimorfismo indica la posibilidad de tomar un objeto (de tipo Animal, por ejemplo), e indicarle que realice la acción de expresarse, esta acción será diferente según el tipo de mamífero del que se trate.

## 6.3. Objetos y Clases

### 6.3.1. Características de los objetos

En este contexto, un objeto de software es una representación de un objeto del mundo real, compuesto de una serie de características y un comportamiento específico. Pero ¿qué es más concretamente un objeto en Programación Orientada a Objetos? Veámoslo.

#### Definición

Un objeto es un conjunto de datos con las operaciones definidas para ellos. Los objetos tienen un estado y un comportamiento.

Por tanto, estudiando los objetos que están presentes en un problema podemos dar con la solución a dicho problema. Los objetos tienen unas características fundamentales que los distinguen:

- **Identidad.** Es la característica que permite diferenciar un objeto de otro. De esta manera, aunque dos objetos sean exactamente iguales en sus atributos, son distintos entre sí. Puede ser una dirección de memoria, el nombre del objeto o cualquier otro elemento que utilice el lenguaje para distinguirlos. Por ejemplo, dos vehículos que hayan salido de la misma cadena de fabricación y sean iguales aparentemente, son distintos porque tienen un código que los identifica.
- **Estado.** El estado de un objeto viene determinado por una serie de parámetros o atributos que lo describen, y los valores de éstos. Por ejemplo, si tenemos un objeto Coche, el estado estaría definido por atributos como Marca, Modelo, Color, Cilindrada, etc.
- **Comportamiento.** Son las acciones que se pueden realizar sobre el objeto. En otras palabras, son los métodos o procedimientos que realiza el objeto. Siguiendo con el ejemplo del objeto Coche, el comportamiento serían acciones como: arrancar(), parar(), acelerar(), frenar(), etc. Definición de clases.

Una clase java se escribe en un fichero con extensión .java que tiene el mismo nombre que la clase. Por ejemplo la clase Vehículo se escribiría en el fichero Vehiculo.java.

Cuando la clase se compila se obtiene un fichero con el mismo nombre que la clase y extensión .class. Ej.: Vehiculo.class.

### Acción

Los identificadores de clase siguen las mismas reglas que otros identificadores de Java (contienen carácter alfanuméricos y especiales, no pueden comenzar por un dígito, no pueden coincidir con una palabra reservada, etc.). Por convenio los identificadores de las clases comienzan por mayúsculas.

#### 6.3.2. Propiedades y métodos de los objetos

Como acabamos de ver todo objeto tiene un estado y un comportamiento. Concretando un poco más, las partes de un objeto son:

- **Campos, Atributos o Propiedades:** Parte del objeto que almacena los datos. También se les denomina Variables Miembro. Estos datos pueden ser de cualquier tipo primitivo (`boolean`, `char`, `int`, `double`, etc) o ser a su vez otro objeto. Por ejemplo, un objeto de la clase `Coche` puede tener un objeto de la clase `Ruedas` (o más concretamente cuatro).
- **Métodos o Funciones Miembro:** Parte del objeto que lleva a cabo las operaciones sobre los atributos definidos para ese objeto.

La idea principal es que el objeto reúne en una sola entidad los datos y las operaciones, y para acceder a los datos privados del objeto debemos utilizar los métodos que hay definidos para ese objeto.

La única forma de manipular la información del objeto es a través de sus métodos. Es decir, si queremos saber el valor de algún atributo, tenemos que utilizar el método que nos muestre el valor de ese atributo. De esta forma, evitamos que métodos externos puedan alterar los datos del objeto de manera inadecuada. Se dice que los datos y los métodos están encapsulados dentro del objeto.

##### 6.3.2.1. ATRIBUTOS

Los atributos representan la información que almacenan los objetos de la clase. Los atributos son declaraciones de variables dentro de la clase.

Se sigue la siguiente sintaxis (los corchetes indican opcionalidad):

```
1 [ámbito] tipo nombreDelAtributo;
2 [ámbito] tipo nombreDelAtributo1, nombreDelAtributo2, ...;
```

donde ...

- **ámbito** permite indicar desde qué clases es accesible el atributo.
- **tipo** indica el tipo de dato del atributo.
- **nombreDelAtributo** es el identificador del atributo.

##### 6.3.2.2. MÉTODOS

Los métodos determinan qué puede hacer un objeto de la clase, es decir, su comportamiento.

Los métodos realizan algún tipo de acción o tarea y, en ocasiones, devuelven un resultado.

Para realizar su trabajo puede ser necesario que pasemos al método cierta información. Por ejemplo, cuando llamamos al método `round` de la clase `Math`, para redondear un número real, debemos indicar al método cual es el número que queremos redondear. A esa información que pasamos a los métodos se le llama **parámetros o argumentos**.

### Ejemplo

```
1 //Al llamar a Math.round, pasamos al método un parámetro
2 int redondeado1 = Math.round(numero);
3 int redondeado2 = Math.round(125.687);
4 ...
5 //Al llamar a Math.pow, pasamos al método dos parámetros
6 int pot1 = Math.pow(a,b);
7 int pot2 = Math.pow(a,6);
8 ...
```

En la definición de un método se distinguen dos partes

- **La cabecera**, en la cual se indica información relevante sobre el método.
- **El cuerpo**, que contiene las instrucciones mediante las cuales el método realiza su tarea.

Para definirlos, se sigue la siguiente sintaxis (los corchetes indican opcionalidad):

## Ejemplo

```

1 public static void main (String[] args)
2 [ámbito] [static] tipoDevuelto nombreDelMetodo ([parámetros]){
3 //Cuerpo del método (instrucciones)
4 ...
5 ...
6 ...
7 }
```

donde...

- **ámbito** permite indicar desde qué clases es accesible el método.
- **static**, cuando aparece, indica que el método es estático.
- **tipoDevuelto** indica el tipo de dato que devuelve el método. La palabra reservada **void** (que no es ningún tipo de dato), indicaría que el método no devuelve nada.
- **nombreDelMetodo** es el identificador del método
- **parámetros** es una lista, separada por comas, de los parámetros que recibe el método. De cada parámetro se indica el **tipo** y un **identificador**.

### 6.3.3. Interacción entre objetos

Dentro de un programa los objetos se comunican llamando unos a otros a sus métodos. Los métodos están dentro de los objetos y describen el comportamiento de un objeto cuando recibe una llamada a uno de sus métodos. En otras palabras, cuando un objeto, `objeto1`, quiere actuar sobre otro, `objeto2`, tiene que ejecutar uno de sus métodos. Entonces se dice que el `objeto2` recibe un mensaje del `objeto1`.

Un mensaje es la acción que realiza un objeto. Un método es la función o procedimiento al que se llama para actuar sobre un objeto.

Los distintos mensajes que puede recibir un objeto o a los que puede responder reciben el nombre de **protocolo** de ese objeto.

El proceso de interacción entre objetos se suele resumir diciendo que se ha "enviado un mensaje" (hecho una petición) a un objeto, y el objeto determina "qué hacer con el mensaje" (ejecuta el código del método). Cuando se ejecuta un programa se producen las siguientes acciones:

- **Creación** de los objetos a medida que se necesitan.
- **Comunicación** entre los objetos mediante el envío de mensajes unos a otros, o el usuario a los objetos.
- **Eliminación** de los objetos cuando no son necesarios para dejar espacio libre en la memoria del computador.

## Recuerda

Los objetos se pueden comunicar entre ellos invocando a los métodos de los otros objetos.

### 6.3.4. Clases

Hasta ahora hemos visto lo que son los objetos. Un programa informático se compone de muchos objetos, algunos de los cuales comparten la misma estructura y comportamiento. Si tuviéramos que definir su estructura y comportamiento del objeto cada vez que queremos crear un objeto, estaríamos utilizando mucho código redundante. Por ello lo que se hace es crear una clase, que es una descripción de un conjunto de objetos que comparten una estructura y un comportamiento común. Y a partir de la clase, se crean tantas "copias" o "instancias" como necesitemos. Esas copias son los objetos de la clase.

## Recuerda

Las clases constan de datos y métodos que resumen las características comunes de un conjunto de objetos. Un programa informático está compuesto por un conjunto de clases, a partir de las cuales se crean objetos que interactúan entre sí.

En otras palabras, una clase es una plantilla o prototipo donde se especifican:

- Los **atributos** comunes a todos los objetos de la clase.

- Los **métodos** que pueden utilizarse para manejar esos objetos.

Para declarar una clase en Java se utiliza la palabra reservada `class`. La declaración de una clase está compuesta por:

- **Cabecera de la clase.** La cabecera es un poco más compleja que como aquí definimos, pero por ahora sólo nos interesa saber que está compuesta por una serie de modificadores, en este caso hemos puesto `public` que indica que es una clase pública a la que pueden acceder otras clases del programa, la palabra reservada `class` y el nombre de la clase.
- **Cuerpo de la clase.** En él se especifican encerrados entre llaves los atributos y los métodos que va a tener la clase.

### Ejemplo

```

1 //Paquete al que pertenece la clase
2 package NombreDePaquete;
3
4 //Paquetes que importa la clase
5 import ...
6
7 ...
8
9 public class NombreDeLaClase {
10 // Atributos de la clase
11 ...
12 ...
13 ...
14 // Métodos de la clase
15 ...
16 ...
17 ...
18 }
```

### Ejemplo

En la unidad anterior ya hemos utilizado clases, aunque aún no sabíamos su significado exacto. Por ejemplo, en los ejemplos de la unidad o en la tarea, estábamos utilizando clases, todas ellas eran clases principales, no tenían ningún atributo y el único método del que disponían era el método `main()`.

### Ejemplo

También es una clase `Math` y su método era `random()`, el que nos permitía usar números aleatorios.

### Ejemplo

El método `main()` se utiliza para indicar que se trata de una clase principal, a partir de la cual va a empezar la ejecución del programa. Este método no aparece si la clase que estamos creando no va a ser la clase principal del programa.

#### 6.3.4.1. ¿QUÉ SIGNIFICA `public class`?

Significa que la clase que se define es pública. Una clase pública es una clase accesible desde otras clases o, dicho de otra forma, que puede ser utilizada por otras clases. Ya hemos dicho que un programa, de alguna manera, consiste en la creación de objetos de distintas clases, que se relacionan entre sí. Lo más común es que las clases que definimos sean públicas y que en cada fichero de extensión `.java` se defina una única clase.

Sin embargo, en ocasiones se definen clases (`A`) que solo van a ser utilizadas por una clase determinada (`B`). En ese caso, decimos que la clase `A` es una clase privada de la clase `B`. Las clases `A` y `B` se definen en el mismo fichero `.java`.

### Acción

En un fichero pueden definirse varias clases pero solo una de ellas puede ser pública. De esta forma, si en un fichero se definen varias clases, una de ellas sería pública y el resto serían clases privadas de la primera, a las que solo ésta tendría acceso.

## 6.4. Utilización de Objetos

Una vez que hemos creado una clase, podemos crear objetos en nuestro programa a partir de esas clases.

Cuando creamos un objeto a partir de una clase se dice que hemos creado una "instancia de la clase". A efectos prácticos, "objeto" e "instancia de clase" son términos similares. Es decir, nos referimos a objetos como instancias cuando queremos hacer hincapié que son de una clase particular.

Los objetos se crean a partir de las clases, y representan casos individuales de éstas.

### Ejemplo

Para entender mejor el concepto entre un objeto y su clase, piensa en un molde de galletas y las galletas. El molde sería la clase, que define las características del objeto, por ejemplo su forma y tamaño. Las galletas creadas a partir de ese molde son los objetos o instancias.

Otro ejemplo, imagina una clase Persona que reúna las características comunes de las personas (`color de pelo, ojos, peso, altura, etc.`) y las acciones que pueden realizar (`crecer, dormir, comer, etc.`). Posteriormente dentro del programa podremos crear un objeto `Trabajador` que esté basado en esa clase Persona. Entonces se dice que el objeto `Trabajador` es una instancia de la clase `Persona`, o que la clase `Persona` es una abstracción del objeto `Trabajador`.

Cualquier objeto instanciado de una clase contiene una copia de todos los atributos definidos en la clase. En otras palabras, lo que estamos haciendo es reservar un espacio en la memoria del ordenador para guardar sus atributos y métodos. Por tanto, cada objeto tiene una zona de almacenamiento propia donde se guarda toda su información, que será distinta a la de cualquier otro objeto. A las variables miembro instanciadas también se les llama variables instancia. De igual forma, a los métodos que manipulan esas variables se les llama métodos instancia.

En el ejemplo del objeto `Trabajador`, las variables instancia serían `color_de_pelo, peso, altura, etc.` Y los métodos instancia serían `crecer(), dormir(), comer()`, etc.

#### 6.4.1. Ciclo de vida de los objetos

Todo programa en Java parte de una única clase, que como hemos comentado se trata de la clase principal.

Esta clase ejecutará el contenido de su método `main()`, el cual será el que utilice las demás clases del programa, cree objetos y lance mensajes a otros objetos.

Las instancias u objetos tienen un tiempo de vida determinado. Cuando un objeto no se va a utilizar más en el programa, es destruido por el recolector de basura para liberar recursos que pueden ser reutilizados por otros objetos.

A la vista de lo anterior, podemos concluir que los objetos tienen un ciclo de vida, en el cual podemos distinguir las siguientes fases:

- **Creación**, donde se hace la reserva de memoria e inicialización de atributos.
- **Manipulación**, que se lleva a cabo cuando se hace uso de los atributos y métodos del objeto.
- **Destrucción**, eliminación del objeto y liberación de recursos.

#### 6.4.2. Declaración

Para la creación de un objeto hay que seguir los siguientes pasos:

- **Declaración**: Definir el tipo de objeto.
- **Instanciación**: Creación del objeto utilizando el operador `new`. Pero ¿en qué consisten estos pasos a nivel de programación en Java? Veamos primero cómo declarar un objeto. Para la definición del tipo de objeto debemos emplear la siguiente instrucción:

```
1 <tipo> nombre_objeto;
```

Donde:

- **tipo** es la clase a partir de la cual se va a crear el objeto, y
- **nombre\_objeto** es el nombre de la variable referencia con la cual nos referiremos al objeto.

Los tipos referenciados o referencias se utilizan para guardar la dirección de los datos en la memoria del ordenador.

Nada más crear una referencia, ésta se encuentra vacía. Cuando una referencia a un objeto no contiene ninguna instancia se dice que es una referencia nula, es decir, que contiene el valor `null`.

Esto quiere decir que la referencia está creada pero que el objeto no está instanciado todavía, por eso la referencia apunta a un objeto inexistente llamado "nulo".

Para entender mejor la declaración de objetos veamos un ejemplo. Cuando veímos los tipos de datos, decíamos que Java proporciona un tipo de dato especial para los textos o cadenas de caracteres que era el tipo de dato `String`. Veímos que realmente este tipo de dato es un tipo referenciado y creábamos una variable mensaje de ese tipo de dato de la siguiente forma:

```
1 String mensaje;
```

Los nombres de la clase empiezan con mayúscula, como `String`, y los nombres de los objetos con minúscula, como `mensaje`, así sabemos qué tipo de elemento utilizando.

Pues bien, `String` es realmente la clase a partir de la cual creamos nuestro objeto llamado `mensaje` 😊.

Si observas, poco se diferencia esta declaración de las declaraciones de variables que hacíamos para los tipos primitivos. Antes decíamos que `mensaje` era una variable del tipo de dato `String`. Ahora realmente vemos que `mensaje` es un objeto de la clase `String`. Pero `mensaje` aún no contiene el objeto porque no ha sido instanciado, veamos cómo hacerlo.

Por tanto, cuando creamos un objeto estamos haciendo uso de una variable que almacena la dirección de ese objeto en memoria. Esa variable es una referencia o un tipo de datos referenciado, porque no contiene el dato si no la posición del dato en la memoria del ordenador.

```
1 String saludo = new String("Bienvenido a Java");
2 String s; //s vale null
3 s = saludo; //asignación de referencias
```

En las instrucciones anteriores, las variables `s` y `saludo` apuntan al mismo objeto de la clase `String`. Esto implica que cualquier modificación en el objeto `saludo` modifica también el objeto al que hace referencia la variable `s`, ya que realmente son el mismo.

#### 6.4.3. Instanciación

Una vez creada la referencia al objeto, debemos crear la instancia u objeto que se va a guardar en esa referencia. Para ello utilizamos la orden `new` con la siguiente sintaxis:

```
1 [<tipo>] nombre_objeto = new <Constructor_de_la_Clase>([<par1>, <par2>, ..., <parN>]);
```

Donde:

- **nombre\_objeto** es el nombre de la variable referencia con la cual nos referiremos al objeto.
- **new** es el operador para crear el objeto.
- **Constructor\_de\_la\_Clase** es un método especial de la clase, que se llama igual que ella, y se encarga de inicializar el objeto, es decir, de dar unos valores iniciales a sus atributos.
- **par1-parN**, son parámetros que puede o no necesitar el constructor para dar los valores iniciales a los atributos del objeto.

Durante la instanciación del objeto, se reserva memoria suficiente para el objeto. De esta tarea se encarga Java y juega un papel muy importante el `recolector de basura`, que se encarga de eliminar de la memoria los objetos no utilizados para que ésta pueda volver a ser utilizada.

De este modo, para instanciar un objeto `String`, haríamos lo siguiente:

```
1 mensaje = new String;
```

Así estaríamos instanciando el objeto `mensaje`. Para ello utilizaríamos el operador `new` y el constructor de la clase `String` a la que pertenece el objeto según la declaración que hemos hecho en el apartado anterior. A continuación utilizamos el constructor, que se llama igual que la clase, `String`.

En el ejemplo anterior el objeto se crearía con la cadena vacía (`""`), si queremos que tenga un contenido debemos utilizar parámetros en el constructor, así:

```
1 mensaje = new String ("El primer programa");
```

Java permite utilizar la clase `String` como si de un tipo de dato primitivo se tratara, por eso no hace falta utilizar el operador `new` para instanciar un objeto de la clase `String` (pero no es lo habitual en el resto de clases).

```
1 mensaje = "El primer programa";
```

La declaración e instanciación de un objeto puede realizarse en la misma instrucción, así:

```
1 String mensaje = new String ("El primer programa");
```

o para la clase `String`:

```
1 String mensaje = "El primer programa";
```

#### 6.4.4. Manipulación

Una vez creado e instanciado el objeto ¿cómo accedemos a su contenido? Para acceder a los atributos y métodos del objeto utilizaremos el nombre del objeto seguido del operador punto ( . ) y el nombre del **atributo** o **método** que queremos utilizar. Cuando utilizamos el operador punto se dice que estamos enviando un mensaje al objeto. La forma general de enviar un mensaje a un objeto es:

```
1 nombre_objeto.mensaje
```

Por ejemplo, para acceder a las variables instancia o atributos se utiliza la siguiente sintaxis:

```
1 nombre_objeto.atributo
```

Y para acceder a los métodos o funciones miembro del objeto se utiliza la sintaxis es:

```
1 nombre_objeto.método([par1, par2, ..., parN])
```

En la sentencia anterior `par1`, `par2`, etc. son los parámetros que utiliza el método. (*Aparece entre corchetes para indicar son opcionales*).

Para entender mejor cómo se manipulan objetos vamos a utilizar un ejemplo. Para ello necesitamos la Biblioteca de Clases Java o API (Application Programming Interface - Interfaz de programación de aplicaciones). Uno de los paquetes de librerías o bibliotecas es `java.awt`. Este paquete contiene clases destinadas a la creación de objetos gráficos e imágenes. Vemos por ejemplo cómo crear un rectángulo.

En primer lugar, instanciamos el objeto utilizando el método constructor, que se llama igual que el objeto, e indicando los parámetros correspondientes a la posición y a las dimensiones del rectángulo:

```
1 Rectangle rect = new Rectangle(50, 50, 150, 150);
```

Una vez instanciado el objeto rectángulo si queremos cambiar el valor de los atributos utilizamos el operador punto. Por ejemplo, para cambiar la dimensión del rectángulo:

```
1 rect.height=100;
2 rect.width=100;
```

O bien podemos utilizar un método para hacer lo anterior:

```
1 rect.setSize(200, 200);
```

A continuación puedes acceder al código del ejemplo:

## Ejemplo

```

1 /*
2 * Muestra como se manipulan objetos en Java
3 */
4 import java.awt.Rectangle;
5 public class Manipular {
6 public static void main(String[] args) {
7 // Instanciamos el objeto rect indicando posicion y dimensiones
8 Rectangle rect = new Rectangle(50, 50, 150, 150);
9 //Consultamos las coordenadas x e y del rectangulo
10 System.out.println("----- Coordenadas esquina superior izqda. -----");
11 System.out.println("\tx = " + rect.x + "\ny = " + rect.y);
12 // Consultamos las dimensiones (altura y anchura) del rectangulo
13 System.out.println("----- Dimensiones -----");
14 System.out.println("\tAlto = " + rect.height);
15 System.out.println("\tAncho = " + rect.width);
16 //Cambiar coordenadas del rectangulo
17 rect.height=100;
18 rect.width=100;
19 rect.setSize(200, 200);
20 System.out.println("\n-- Nuevos valores de los atributos --");
21 System.out.println("\tx = " + rect.x + "\ny = " + rect.y);
22 System.out.println("\tAlto = " + rect.height);
23 System.out.println("\tAncho = " + rect.width);
24 }
25 }
```

### 6.4.5. Destrucción de objetos y liberación de memoria

Cuando un objeto deja de ser utilizado, es necesario liberar el espacio de memoria y otros recursos que poseía para poder ser reutilizados por el programa. A esta acción se le denomina destrucción del objeto.

En Java la destrucción de objetos corre a cargo del **recolector de basura (garbage collector)**. Es un sistema de destrucción automática de objetos que ya no son utilizados. Lo que se hace es liberar una zona de memoria que había sido reservada previamente mediante el operador `new`. Esto evita que los programadores tengan que preocuparse de realizar la liberación de memoria.

El recolector de basura se ejecuta en modo segundo plano y de manera muy eficiente para no afectar a la velocidad del programa que se está ejecutando. Lo que hace es que periódicamente va buscando objetos que ya no son referenciados, y cuando encuentra alguno lo marca para ser eliminado.

Después los elimina en el momento que considera oportuno.

Justo antes de que un objeto sea eliminado por el recolector de basura, se ejecuta su método `finalize()`. Si queremos forzar que se ejecute el proceso de finalización de todos los objetos del programa podemos utilizar el método `runFinalization()` de la clase `System`. La clase `System` forma parte de la Biblioteca de Clases de Java y contiene diversas clases para la entrada y salida de información, acceso a variables de entorno del programa y otros métodos de diversa utilidad. Para forzar el proceso de finalización ejecutaríamos:

```
1 System.runFinalization();
```

## 6.5. Utilización de Métodos

Los métodos, junto con los atributos, forman parte de la estructura interna de un objeto. Los métodos contienen la declaración de variables locales y las operaciones que se pueden realizar para el objeto, y que son ejecutadas cuando el método es invocado. Se definen en el cuerpo de la clase y posteriormente son instanciados para convertirse en métodos instancia de un objeto.

Para utilizar los métodos adecuadamente es conveniente conocer la estructura básica de que disponen.

Al igual que las clases, los métodos están compuestos por una cabecera y un cuerpo. La cabecera también tiene modificadores, en este caso hemos utilizado `public` para indicar que el método es público, lo cual quiere decir que le pueden enviar mensajes no sólo los métodos del objeto sino los métodos de cualquier otro objeto externo.

Dentro de un método nos encontramos el cuerpo del método que contiene el código de la acción a realizar. Las acciones que un método puede realizar son:

- **Iniciar** los atributos del objeto
- **Consultar** los valores de los atributos
- **Modificar** los valores de los atributos
- **Llamar a otros métodos**, del mismo del objeto o de objetos externos

### 6.5.1. Parámetros y valores devueltos

Los métodos se pueden utilizar tanto para consultar información sobre el objeto como para modificar su estado. La información consultada del objeto se devuelve a través de lo que se conoce como valor de retorno, y la modificación del estado del objeto, o sea, de sus atributos, se hace mediante la lista de parámetros. En general, la lista de parámetros de un método se puede declarar de dos formas diferentes:

- **Por valor.** El valor de los parámetros no se devuelve al finalizar el método, es decir, cualquier modificación que se haga en los parámetros no tendrá efecto una vez se salga del método. Esto es así porque cuando se llama al método desde cualquier parte del programa, dicho método recibe una copia de los argumentos, por tanto cualquier modificación que haga será sobre la copia, no sobre las variables originales.
- **Por referencia.** La modificación en los valores de los parámetros sí tienen efecto tras la finalización del método. Cuando pasamos una variable a un método por referencia lo que estamos haciendo es pasar la dirección del dato en memoria, por tanto cualquier cambio en el dato seguirá modificado una vez salgamos del método.

#### Acción

En el lenguaje Java, todas las variables se pasan por valor, excepto los objetos que se pasan por referencia.

En Java, la declaración de un método tiene dos restricciones:

- **Un método siempre tiene que devolver un valor (no hay valor por defecto).** Este valor de retorno es el valor que devuelve el método cuando termina de ejecutarse, al método o programa que lo llamó. Puede ser un tipo primitivo, un tipo referenciado o bien el tipo `void`, que indica que el método no devuelve ningún valor.
- **Un método tiene un número fijo de argumentos.** Los argumentos son variables a través de las cuales se pasa información al método desde el lugar del que se llame, para que éste pueda utilizar dichos valores durante su ejecución. Los argumentos reciben el nombre de parámetros cuando aparecen en la declaración del método.

#### Recuerda

El valor de retorno es la información que devuelve un método tras su ejecución.

Según hemos visto en el apartado anterior, la cabecera de un método se declara como sigue:

```
1 public tipo_de_dato_devuelto nombreMetodo (lista_de_parametros);
```

Como vemos, el tipo de dato devuelto aparece después del modificador `public` y se corresponde con el valor de retorno.

La lista de parámetros aparece al final de la cabecera del método, justo después del nombre, encerrados entre signos de paréntesis y separados por comas. Se debe indicar el tipo de dato de cada parámetro así:

```
1 (tipo_parametro1 nombre_parametro1, ..., tipo_parametroN nombre_parametroN)
```

#### Acción

Cuando se llame al método, se deberá utilizar el nombre del método, seguido de los argumentos que deben coincidir con la lista de parámetros.

La lista de argumentos en la llamada a un método debe coincidir en número, tipo y orden con los parámetros del método, ya que de lo contrario se produciría un error de sintaxis.

### 6.5.2. Constructores

¿Recuerdas cuando hablábamos de la creación e instanciación de un objeto? Decíamos que utilizábamos el operador `new` seguido del nombre de la clase y una pareja de abrir-cerrar paréntesis.

Además, el nombre de la clase era realmente el constructor de la misma, y lo definíamos como un método especial que sirve para inicializar valores. En este apartado vamos a ver un poco más sobre los constructores.

Un constructor es un método especial con el mismo nombre de la clase y que no devuelve ningún valor tras su ejecución.

Cuando creamos un objeto debemos instanciarlo utilizando el constructor de la clase. Veamos la clase `Date` proporcionada por la Biblioteca de Clases de Java. Si queremos instanciar un objeto a partir de la clase `Date` tan sólo tendremos que utilizar el constructor seguido de una pareja de abrir-cerrar paréntesis:

```
1 Date fecha = new Date();
```

Con la anterior instrucción estamos creando un objeto fecha de tipo `Date`, que contendrá la fecha y hora actual del sistema.

La estructura de los constructores es similar a la de cualquier método, salvo que no tiene tipo de dato devuelto porque no devuelve ningún valor. Está formada por una cabecera y un cuerpo, que contiene la inicialización de atributos y resto de instrucciones del constructor.

El método constructor tiene las siguientes particularidades:

- **El constructor es invocado** automáticamente **en la creación** de un objeto, **y sólo esa vez**.
- **Los constructores no empiezan con minúscula**, como el resto de los métodos, ya que se llaman igual que la clase y las clases empiezan con letra mayúscula.
- **Puede haber varios** constructores para una clase.
- Como cualquier método, **el constructor puede tener parámetros** para definir qué valores dar a los atributos del objeto.
- **El constructor por defecto es aquél que no tiene argumentos o parámetros**. Cuando creamos un objeto llamando al nombre de la clase sin argumentos, estamos utilizando el constructor por defecto.
- **Es necesario que toda clase tenga al menos un constructor**. Si no definimos constructores para una clase, y sólo en ese caso, el compilador crea un constructor por defecto vacío, que inicializa los atributos a sus valores por defecto, según del tipo que sean: `0` para los tipos numéricos, `false` para los `boolean` y `null` para los tipo carácter y las referencias. Dicho constructor lo que hace es llamar al constructor sin argumentos de la superclase (clase de la cual hereda); si la superclase no tiene constructor sin argumentos se produce un error de compilación.

### Atención

Cuando definimos constructores personalizados, el constructor por defecto deja de existir, y si no definimos nosotros un constructor sin argumentos cuando intentemos utilizar el constructor por defecto nos dará un error de compilación.

#### 6.5.3. El operador `this`

Los constructores y métodos de un objeto suelen utilizar el operador `this`. Este operador sirve para referirse a los atributos de un objeto cuando estamos dentro de él. Sobre todo se utiliza cuando existe ambigüedad entre el nombre de un parámetro y el nombre de un atributo, entonces en lugar del nombre del atributo solamente escribiremos `this.nombre_atributo`, y así no habrá duda de a qué elemento nos estamos refiriendo.

#### 6.5.4. Métodos estáticos

Cuando trabajábamos con cadenas de caracteres utilizando la clase `String`, veíamos las operaciones que podíamos hacer con ellas: obtener longitud, comparar dos cadenas de caracteres, cambiar a mayúsculas o minúsculas, etc. Pues bien, sin saberlo estábamos utilizando métodos estáticos definidos por Java para la clase `String`. Pero ¿qué son los métodos estáticos? Veámoslo.

Los métodos estáticos son aquellos métodos definidos para una clase que se pueden usar directamente, sin necesidad de crear un objeto de dicha clase. También se llaman métodos de clase.

Para llamar a un método estático utilizaremos:

- **El nombre del método**, si lo llamamos desde la misma clase en la que se encuentra definido.
- **El nombre de la clase**, seguido por el operador punto (`.`) más el nombre del método estático, si lo llamamos desde una clase distinta a la que se encuentra definido:

```
1 Nombre_clase.nombre_metodo_estatico
```

- **El nombre del objeto**, seguido por el operador punto (`.`) más el nombre del método estático. Utilizaremos esta forma cuando tengamos un objeto instanciado de la clase en la que se encuentra definido el método estático, y no podamos utilizar la anterior:

```
1 nombre_objeto.nombre_metodo_no_estatico
```

Los métodos estáticos no afectan al estado de los objetos instanciados de la clase (variables instancia), y suelen utilizarse para realizar operaciones comunes a todos los objetos de la clase. Por ejemplo, si necesitamos contar el número de objetos instanciados de una clase, podríamos utilizar un método estático que fuera incrementando el valor de una variable entera de la clase conforme se van creando los objetos.

En la Biblioteca de Clases de Java existen muchas clases que contienen métodos estáticos. Pensemos en las clases que ya hemos utilizado en unidades anteriores, como hemos comentado la clase `String` con todas las operaciones que podíamos hacer con ella y con los objetos instanciados a partir de ella. O bien la clase `Math` para la conversión de tipos de datos. Todos ellos son métodos estáticos que la API de Java define para esas clases. Lo importante es que tengamos en cuenta que al tratarse de métodos estáticos, para utilizarlos no necesitamos crear un objeto de dichas clases.

Fijémonos en esta secuencia de instrucciones

```
1 //Creamos dos círculos de radio 100 en distintas posiciones
2 Circulo c1 = new Circulo(50,50,100);
3 Circulo c2 = new Circulo(80,80,100);
4 ...
5 //Aumentamos el radio del primer círculo a 200
6 c1.setRadio(200);
```

y en esta otra

```
1 System.out.println(Math.sqrt(4));
```

En el primer ejemplo, `.setRadio(200)` va precedido por un objeto. La variable `c1` es un objeto de la clase Círculo, por tanto, la instrucción está modificando el radio de un círculo concreto, el que se encuentra en la posición (50,50). El método `setRadio` es un método no estático. Los métodos no estáticos actúan siempre sobre algún objeto (el que figura a la izquierda del punto).

En el segundo ejemplo, en cambio, a la izquierda de `.sqrt(4)` no se ha puesto el nombre de un objeto, sino el de una clase, la clase `Math`. El método `sqrt` no está actuando sobre un objeto concreto: no tiene sentido hacerlo, solo pretendemos calcular la raíz cuadrada de 4. `Sqrt` es un método estático. Los métodos estáticos se usan poniendo delante del punto el nombre de la clase en que se encuentran definidos.

## 6.6. Librerías de Objetos (Paquetes)

Conforme nuestros programas se van haciendo más grandes, el número de clases va creciendo. Meter todas las clases en único directorio no ayuda a que estén bien organizadas, lo mejor es hacer grupos de clases, de forma que todas las clases que estén relacionadas o traten sobre un mismo tema estén en el mismo grupo.

Un **paquete** de clases es una agrupación de clases que consideramos que están relacionadas entre sí o tratan de un tema común.

### Recuerda

Las clases de un mismo paquete tienen un acceso privilegiado a los atributos y métodos de otras clases de dicho paquete. Es por ello por lo que se considera que los paquetes son también, en cierto modo, unidades de encapsulación y ocultación de información.

Java nos ayuda a organizar las clases en paquetes. En cada fichero `.java` que hagamos, al principio, podemos indicar a qué paquete pertenece la clase que hagamos en ese fichero.

Los paquetes se declaran utilizando la palabra clave `package` seguida del nombre del paquete.

Para establecer el paquete al que pertenece una clase hay que poner una sentencia de declaración como la siguiente al principio de la clase:

```
1 package nombre_de_Paquete;
```

Por ejemplo, si decidimos agrupar en un paquete `ejemplos` un programa llamado `Bienvenida`, pondríamos en nuestro fichero `Bienvenida.java` lo siguiente:

```
1 package ejemplos;
2
3 public class Bienvenida {
4 [...]
5 }
```

El código es exactamente igual que como hemos venido haciendo hasta ahora, solamente hemos añadido la línea `package ejemplos;` al principio.

### 6.6.1. Sentencia `import`

Cuando queremos utilizar una clase que está en un paquete distinto a la clase que estamos utilizando, se suele utilizar la sentencia `import`. Por ejemplo, si queremos utilizar la clase `Scanner` que está en el paquete `java.util` de la Biblioteca de Clases de Java, tendremos que utilizar esta sentencia:

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

Se pueden importar todas las clases de un paquete, así:

```
1 import java.awt.*;
```

Esta sentencia debe aparecer al principio de la clase, justo después de la sentencia `package`, si ésta existiese.

También podemos utilizar la clase sin sentencia `import`, en cuyo caso cada vez que queramos usarla debemos indicar su ruta completa:

```
1 java.util.Scanner teclado = new java.util.Scanner (System.in);
```

Hasta aquí todo correcto. Sin embargo, al trabajar con paquetes, Java nos obliga a organizar los directorios, compilar y ejecutar de cierta forma para que todo funcione adecuadamente.

### 6.6.2. Librerías Java

Cuando descargamos el entorno de compilación y ejecución de Java, obtenemos la API de Java. Como ya sabemos, se trata de un conjunto de bibliotecas que nos proporciona paquetes de clases útiles para nuestros programas. Utilizar las clases y métodos de la Biblioteca de Java nos va ayudar a reducir el tiempo de desarrollo considerablemente, por lo que es importante que aprendamos a consultarla y conozcamos las clases más utilizadas.

#### Ejemplo

```
1 import java.lang.System; // Se importa la clase System.
2 import java.awt.*; // Se importa todas las clases del paquete awt;
```

Los paquetes más importantes que ofrece el lenguaje Java son:

| Paquete o librería | Descripción                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>java.io</b>     | Contiene las clases que gestionan la entrada y salida, ya sea para manipular ficheros, leer o escribir en pantalla, en memoria, etc. Este paquete contiene por ejemplo la clase <code>BufferedReader</code> que se utiliza para la entrada por teclado.                                                                                                                                                                                                                                                                                                                                                                                                |
| <b>java.lang</b>   | Contiene las clases básicas del lenguaje. Este paquete no es necesario importarlo, ya que es importado automáticamente por el entorno de ejecución. En este paquete se encuentra la clase <code>Object</code> , que sirve como raíz para la jerarquía de clases de Java, o la clase <code>System</code> que ya hemos utilizado en algunos ejemplos y que representa al sistema en el que se está ejecutando la aplicación. También podemos encontrar en este paquete las clases que "envuelven" los tipos primitivos de datos. Lo que proporciona una serie de métodos para cada tipo de dato de utilidad, como por ejemplo las conversiones de datos. |
| <b>java.util</b>   | Biblioteca de clases de utilidad general para el programador. Este paquete contiene por ejemplo la clase <code>Scanner</code> utilizada para la entrada por teclado de diferentes tipos de datos, la clase <code>Date</code> , para el tratamiento de fechas, etc.                                                                                                                                                                                                                                                                                                                                                                                     |
| <b>java.math</b>   | Contiene herramientas para manipulaciones matemáticas.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <b>java.awt</b>    | Incluye las clases relacionadas con la construcción de interfaces de usuario, es decir, las que nos permiten construir ventanas, cajas de texto, botones, etc. Algunas de las clases que podemos encontrar en este paquete son <code>Button</code> , <code>TextField</code> , <code>Frame</code> , <code>Label</code> , etc.                                                                                                                                                                                                                                                                                                                           |

| Paquete o<br>librería | Descripción                                                                                                                                                                                                                                                                |
|-----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>java.swing</b>     | Contiene otro conjunto de clases para la construcción de interfaces avanzadas de usuario. Los componentes que se engloban dentro de este paquete se denominan componentes Swing, y suponen una alternativa mucho más potente que AWT para construir interfaces de usuario. |
| <b>java.net</b>       | Conjunto de clases para la programación en la red local e Internet.                                                                                                                                                                                                        |
| <b>java.sql</b>       | Contiene las clases necesarias para programar en Java el acceso a las bases de datos.                                                                                                                                                                                      |
| <b>java.security</b>  | Biblioteca de clases para implementar mecanismos de seguridad.                                                                                                                                                                                                             |

Como se puede comprobar Java ofrece una completa jerarquía de clases organizadas a través de paquetes.

## 6.7. Cadenas de caracteres. La clase String

### 6.7.1. Cadenas de caracteres

Hasta ahora hemos utilizado literales de cadenas de caracteres que, como sabemos, se ponen entre comillas dobles, como en la siguiente expresión

```
1 System.out.println("Hola");
```

Para almacenar cadenas de caracteres en variables se utiliza la clase `String`. `String` se encuentra definida en el paquete `java.lang`. *Recordemos que no es necesario importar este paquete para utilizar sus clases.*

La forma de `String` es la siguiente:

```
1 String variable = new String("texto");
```

Ejemplo:

```
1 String nombre = new String("Javier");
2 System.out.println("Mi nombre es " + nombre);
```

**Sin embargo**, debido a que es una clase que se utiliza ampliamente en los programas, Java permite una forma abreviada de crear objetos `String`:

```
1 String nombreVariable = "texto";
```

Ejemplo:

```
1 String nombre = "Javier";
2 System.out.println("Mi nombre es " + nombre);
```

### 6.7.2. Leer cadenas desde teclado

#### 6.7.2.1. CLASE Scanner

Para leer cadenas de caracteres desde teclado podemos utilizar la clase `Scanner`. Ésta dispone de dos métodos para leer cadenas:

- `next()` : Lee desde la entrada estándar (teclado) una secuencia de caracteres hasta encontrar un delimitador (un espacio). Devuelve un `String`.
- `nextLine()` : Lee desde la entrada estándar (teclado) una secuencia de caracteres hasta encontrar un salto de línea. Devuelve un `String`.

Ejemplo:

```
1 Scanner tec = new Scanner(System.in);
2 //De lo que introduce el usuario, lee la 1º palabra.
3 String nombre = tec.next();
4 //Lee lo que introduce el usuario hasta que pulsa intro.
5 String nombreCompleto = tec.nextLine();
```

#### 6.7.2.2. EJEMPLOS DE LA UD01 PERO UTILIZANDO `Scanner` (COMPATIBLE CON LOS IDE'S)

A continuación vamos a ver los mismos ejemplos de la UD01, pero utilizando la clase `Scanner` que si es compatible con los IDE's. Para poder usar la clase `Scanner` necesitamos importar el paquete: `java.util.Scanner`.

```

1 import java.util.Scanner;
2
3 public class EjemploUD02 {
4
5 public static void main(String[] args) {
6
7 Scanner teclado = new Scanner(System.in);
8
9 //Introducir texto desde teclado
10 String texto;
11 System.out.print("Introduce un texto: ");
12 texto = teclado.nextLine();
13 System.out.println("El texto introducido es: "+ texto);
14
15 //Introducir un número entero desde teclado
16 String texto2;
17 int entero2;
18 System.out.print("Introduce un número: ");
19 texto2 = teclado.nextLine();
20 entero2 = Integer.parseInt(texto2);
21 System.out.println("El número introducido es:"+entero2);
22
23 //Introducir un número decimal desde teclado
24 String texto3;
25 double doble3;
26 System.out.print("Introduce un número decimal: ");
27 texto3 = teclado.nextLine();
28 doble3 = Double.parseDouble(texto3); // convertimos texto a doble
29 System.out.println("Número decimal introducido es: "+doble3);
30 }
31 }
```

#### 6.7.3. La clase `String`

Además de permitir almacenar cadenas de caracteres, `String` tiene métodos para realizar cálculos u operaciones con ellas.

Así por ejemplo, la clase tiene un método `toUpperCase()` que devuelve el `String` convertido a mayúsculas. El siguiente ejemplo ilustra su uso:

```

1 String nombre = "Javier";
2 System.out.println(nombre.toUpperCase()); // Se muestra JAVIER por pantalla
```

Accede a la documentación en línea de Java y estudia los siguientes métodos de la clase:

- `charAt`
- `indexOf`
- `substring`
- `toLowerCase`
- `trim`

#### 6.7.4. `printf` o `format`

El método `printf()` o `format()` (son sinónimos) utilizan unos códigos de conversión para indicar si el contenido a mostrar de qué tipo es. Estos códigos se caracterizan porque llevan delante el símbolo %, algunos de ellos son:

- `%c` : Escribe un carácter.
- `%s` : Escribe una cadena de texto.
- `%d` : Escribe un entero.
- `%f` : Escribe un número en punto flotante.
- `%e` : Escribe un número en punto flotante en notación científica.

Por ejemplo, si queremos escribir el número float `12345.1684` con el punto de los miles y sólo dos cifras decimales la orden sería:

```
1 System.out.printf("%,.2f\n", 12345.1684);
```

Esta orden mostraría el número `12.345,17` por pantalla.

Otro ejemplo seria:

```
1 System.out.format("El valor de la variable float es" +
2 "%f, mientras que el valor del entero es %d" +
3 "y el string contiene %s", variableFloat, variableInt, variableString);
```

Puedes investigar más sobre `printf` o `format` en este [enlace](#)

#### 6.7.5. Salida de error

La salida de error está representada por el objeto `System.err`. No parece muy útil utilizar `out` y `err` si su destino es la misma pantalla, o al menos en el caso de la consola del sistema donde las dos salidas son representadas con el mismo color y no notamos diferencia alguna. En cambio en la consola de varios entornos integrados de desarrollo como NetBeans o Eclipse la salida de `err` se ve en un color diferente. Teniendo el siguiente código:

```
1 System.out.println("Salida estándar por pantalla");
2 System.err.println("Salida de error por pantalla");
```

Tanto NetBeans, Eclipse como IntelliJ Idea mostraran el mensaje `err` en color rojo.

### 6.8. Ejemplo UD02

#### 6.8.1. Clase Pajaro

Vamos a ilustrar mediante un ejemplo la utilización de objetos y métodos, así como el uso de parámetros y el operador `this`. Aunque la creación de clases la veremos en las siguientes unidades, en este ejercicio creamos una pequeña clase para que podamos instanciar el objeto con el que vamos a trabajar.

Las clases se suelen representar como un rectángulo, y dentro de él se sitúan los atributos y los métodos de dicha clase.

En la imagen, la clase `Pajaro` está compuesta por tres atributos, uno de ellos el `nombre` y otros dos que indican la posición del ave, `posX` y `posY`. Tiene tres métodos constructores y un método `volar()`. Como sabemos, los métodos constructores reciben el mismo nombre de la clase, y puede haber varios para una misma clase, dentro de ella se diferencian unos de otros por los parámetros que utilizan.

#### Ejunciado:

Dada una clase principal llamada `Pajaro`, se definen los atributos y métodos que aparecen en la imagen. Los métodos realizan las siguientes acciones:

```
classDiagram
 class Pajaro{
 -String nombre
 -int posX
 -int posY
 +Pajaro()
 +Pajaro(String nombre)
 +Pajaro(String nombre, int posX, int posY)
 +double volar(int posX, int posY)
 }
```

- `Pajaro()` . Constructor por defecto. En este caso, el constructor por defecto no contiene ninguna instrucción, ya que Java inicializa de forma automática las variables miembro, si no le damos ningún valor.
- `Pajaro(String nombre)` . Constructor que recibe como argumentos una cadena de texto (el nombre del pájaro).
- `Pajaro(String nombre, int posX, int posY)` . Constructor que recibe como argumentos una cadena de texto y dos enteros para inicializar el valor de los atributos.
- `double volar(int posX, int posY)` . Método que recibe como argumentos dos enteros: `posX` y `posY`, y devuelve un valor de tipo `double` como resultado, usando la palabra clave `return`. El valor devuelto es el resultado de aplicar un desplazamiento de acuerdo con la siguiente fórmula:

$$\text{desplazamiento} = \sqrt{\text{posX}^2 + \text{posY}^2}$$

Diseña un programa que utilice la clase `Pajaro`, cree una instancia de dicha clase y ejecute sus métodos.

Lo primero que debemos hacer es crear la clase `Pajaro`, con sus métodos y atributos. De acuerdo con los datos que tenemos, el código de la clase sería el siguiente:

```

1 public class Pajaro {
2 //atributos/variables
3 private String nombre;
4 private int posX;
5 private int posY;
6 //constructores
7 public Pajaro() {
8 }
9 public Pajaro(String nombre) {
10 this.nombre = nombre;
11 }
12 public Pajaro(String nombre, int posX, int posY) {
13 this.nombre = nombre;
14 this.posX = posX;
15 this.posY = posY;
16 }
17
18 //metodos
19 public double volar(int posX, int posY) {
20 double desplazamiento = Math.sqrt((posX * posX) + (posY * posY));
21 //desplazamiento=Math.sqrt(Math.pow(posX,2)+Math.pow(posY,2));
22 this.posX = posX;
23 this.posY = posY;
24 return desplazamiento;
25 }
26 //método main()
27 [...]
28 }
```

Debemos tener en cuenta que se trata de una clase principal, lo cual quiere decir que debe contener un método `main()` dentro de ella. En el método `main()` vamos a situar el código de nuestro programa. El ejercicio dice que tenemos que crear una instancia de la clase y ejecutar sus métodos, entre los que están el constructor y el método `volar()`.

También es conveniente imprimir el resultado de ejecutar el método `volar()`. Por tanto, lo que haría el programa sería:

- Crear un objeto de la clase e inicializarlo.
- Invocar al método volar.
- Imprimir por pantalla la distancia recorrida.

Para inicializar el objeto utilizaremos el constructor con parámetros, después ejecutaremos el método `volar()` del objeto creado y finalmente imprimiremos el valor que nos devuelve el método.

Luego crearemos otro `pajaro2` usando el constructor por defecto (sin parámetros). Le asignaremos el nombre y la posición manualmente, y calcularemos su desplazamiento llamando al método, pero usando los atributos del objeto (`pajaro2.posX` y `pajaro2.posY`) en lugar de constantes. El código del método `main()` quedaría como sigue:

```

1 public static void main(String[] args) {
2 //creamos el objeto con parámetros
3 Pajaro pajaro1 = new Pajaro("WoodPecker", 50, 50);
4 double d1 = pajaro1.volar(50, 50);
5 System.out.println("El desplazamiento de " + pajaro1.nombre + " ha sido " + d1);
6
7 Pajaro pajaro2 = new Pajaro();
8 //damos nombre y cambiamos la posición de "Piolin" a mano
9 pajaro2.nombre="Piolin";
10 pajaro2.posX=30;
11 pajaro2.posY=30;
12 double d2 = pajaro2.volar(pajaro2.posX, pajaro2.posY);
13 System.out.println("El desplazamiento de " + pajaro2.nombre + " ha sido " + d2);
14 }
```

Si ejecutamos nuestro programa el resultado sería el siguiente:

```

1 El desplazamiento de WoodPecker ha sido 70.71067811865476
2 El desplazamiento de Piolin ha sido 42.42640687119285
```

### 6.8.2. Clase String

#### Ejemplo:

```

1 package UD02;
2
3 import java.util.Scanner;
4
5 public class EjemploUD02 {
6
7 public static void main(String[] args) {
8
9 Scanner teclado = new Scanner(System.in);
10
11 //Introducir texto desde teclado
12 String texto;
13 System.out.print("Introduce un texto: ");
14 texto = teclado.nextLine();
15 System.out.println("El texto introducido es: " + texto);
16
17 //Introducir un número entero desde teclado
18 String texto2;
19 int entero2;
20 System.out.print("Introduce un número: ");
21 texto2 = teclado.nextLine();
22 entero2 = Integer.parseInt(texto2);
23 System.out.println("El número introducido es:" + entero2);
24
25 //Introducir un número decimal desde teclado
26 String texto3;
27 double doble3;
28 System.out.print("Introduce un número decimal: ");
29 texto3 = teclado.nextLine();
30 doble3 = Double.parseDouble(texto3); // convertimos texto a doble
31 System.out.println("Número decimal introducido es: " + doble3);
32
33 System.out.println("La clase String");
34 String nombre = "Javier "; //Observa que hay un espacio final
35 System.out.println(nombre.toUpperCase()); //JAVIER
36 System.out.println(nombre.charAt(4)); //e
37 System.out.println(nombre.indexOf("i")); //3
38 System.out.println(nombre.substring(0, 3)); //Javi
39 System.out.println(nombre.toLowerCase()); //javier
40 System.out.println(nombre.trim()); //Javier sin espacios finales
41 System.out.printf("%.2f\n", 12345.1684);
42 nombre.toUpperCase().substring(0,3).indexOf("I"); //3
43 System.out.format("El valor de la variable float es %f"
44 + ", mientras que el valor del entero es %d"
45 + " y el string contiene %s", doble3, entero2, texto);
46
47 System.err.println("Salida de error por pantalla");
48 }
49 }
```

## 6.9. Ejemplos UD02

[EjemploUD02.java](#)

[Pajaro.java](#)

## 6.10. Píldoras informáticas relacionadas

⌚8 de enero de 2026

## 7. 3.2 Ejercicios de la UD02

---

### 7.1. Actividades

1. (Temperatura) Crear una clase llamada Temperatura con dos métodos:

- `celsiusToFahrenheit`. Convierte grados *Celsius* a *Fahrenheit*.  

$$F = (1,8 * C) + 32$$
- `fahrenheitToCelsius`. Convierte grados *Fahrenheit* a *Celsius*.  

$$C = \frac{F - 32}{1,8}$$

2. (Moto) A partir de la siguiente clase:

```

1 public class Moto {
2
3 private int velocidad;
4
5 public Moto() {
6 velocidad=0;
7 }
8 }
```

Añade los siguientes métodos:

- `int getVelocidad()`. Devuelve la velocidad del objeto moto.
- `void acelera(int mas)`. Permite aumentar la velocidad del objeto moto.
- `void frena(int menos)`. Permite reducir la velocidad del objeto moto.

3. (Rebajas) Crea una clase `Rebajas` con un método `descubrePorcentaje()` que descubra el descuento aplicado en un producto. El método recibe el precio original del producto y el rebajado y devuelve el porcentaje aplicado. Podemos calcular el descuento realizando la operación:  $\frac{\text{precioOriginal} - \text{precioRebajado}}{\text{precioOriginal}} \times 100$

4. (Finanzas) Realiza una clase `Finanzas` que convierta dólares a euros y viceversa. Codifica los métodos `dolaresToEuros` y `eurosToDolares`. Prueba que dicha clase funciona correctamente haciendo conversiones entre euros y dólares. La clase tiene que tener:

- Un constructor `Finanzas()` por defecto el cual establece el cambio Dólar-Euro en 1.36.
- Un constructor `Finanzas(double cambio)`, el cual permitirá configurar el cambio Dólar-euro a una cantidad personalizada.

5. (MiNumero) Realiza una clase `MiNumero` que proporcione el doble, triple y cuádruple de un número proporcionado en su constructor (realiza un método para `doble`, otro para `triple` y otro para `cuadruple`).

(Opcional, no hay puntos) Haz que la clase tenga un método `main` y comprueba los distintos métodos.

6. (Numero) Realiza una clase `Numero` que almacene un número entero y tenga las siguientes características:

- Constructor por defecto que inicializa a 0 el número interno.
- Constructor que inicializa el número interno.
- Método `anade` que permite sumarle un número al valor interno.
- Método `resta` que resta un número al valor interno.
- Método `getValor`. Devuelve el valor interno.
- Método `getDoble`. Devuelve el doble del valor interno.
- Método `getTriple`. Devuelve el triple del valor interno.
- Método `setNumero`. Inicializa de nuevo el valor interno.

7. (Peso) Crea la clase `Peso`, la cual tendrá las siguientes características:

- Deberá tener un atributo donde se almacene el peso de un objeto en kilogramos. En el constructor se le pasará el peso y la medida en la que se ha tomado ("Lb" para libras, "Li" para lingotes, "Oz" para onzas, "P" para peniques, "K" para kilos, "G" para gramos y "Q" para quintales).
- Deberá de tener los siguientes métodos:
  - `getLibras`. Devuelve el peso en libras.
  - `getLingotes`. Devuelve el peso en lingotes.
  - `getPeso`. Devuelve el peso en la medida que se pase como parámetro ("Lb" para libras, "Li" para lingotes, "Oz" para onzas, "P" para peniques, "K" para kilos, "G" para gramos y "Q" para quintales).
- Para la realización del ejercicio toma como referencia los siguientes datos:

- 1 Libra = 16 onzas = 453 gramos.
- 1 Lingote = 32,17 libras = 14,59 kg.
- 1 Onza = 0,0625 libras = 28,35 gramos.
- 1 Penique = 0,05 onzas = 1,55 gramos.
- 1 Quintal = 100 libras = 43,3 kg.

(Opcional, no hay puntos) Crea además un método `main` para testear y verificar los métodos de esta clase.

8. (Millas) Crea una clase con un método `millasAMetros()` que toma como parámetro de entrada un valor en millas marinas y las convierte a metros. Una vez tengas este método escribe otro `millasAKilometros()` que realice la misma conversión, pero esta vez exprese el resultado en kilómetros.

*Nota: 1 milla marina equivale a 1852 metros.*

9. (Coche) Crea la clase `Coche` con dos constructores. Uno no toma parámetros y el otro sí. Los dos constructores inicializarán los atributos `marca` y `modelo` de la clase. El constructor por defecto (sin parámetros) crea el coche "Ford" modelo "C-MAX".

(Opcional, no hay puntos) Crea dos objetos (cada objeto llama a un constructor distinto) y verifica que todo funciona correctamente.

10. (Consumo) Implementa una clase `Consumo`, la cual forma parte del "ordenador de a bordo" de un coche y tiene las siguientes características:

- Atributos:
  - `kilometros`.
  - `litros`. Litros de combustible consumido.
  - `vmed`. Velocidad media.
  - `pgas`. Precio de la gasolina.
- Métodos:
  - `getTiempo`. Indicará el tiempo empleado en realizar el viaje.
  - `consumoMedio`. Consumo medio del vehículo (en litros cada 100 kilómetros).
  - `consumoEuros`. Consumo medio del vehículo (en euros cada 100 kilómetros).

No olvides crear un constructor para la clase que establezca el valor de los atributos. Elige el tipo de datos más apropiado para cada atributo.

11. (ConsumoModificadores) Para la clase anterior implementa los siguientes métodos, los cuales podrán modificar los valores de los atributos de la clase:

- `setKilometros`
- `setLitros`
- `setVmed`
- `setPgas`

12. (Restaurante) Un restaurante cuya especialidad son las patatas con carne nos pide diseñar un método (`calcularClientes`) con el que se pueda saber cuántos clientes pueden atender con la materia prima que tienen en el almacén. El método recibe la cantidad de patatas y carne en kilos y devuelve el número de clientes que puede atender el restaurante teniendo en cuenta que por cada tres personas, utilizan un dos kilos de patatas y un kilo de carne.

13. (RestauranteClase) Modifica el programa anterior creando una clase que permita almacenar los kilos de patatas y carne del restaurante. Implementa los siguientes métodos:

- `public void addCarne(int x)`. Añade `x` kilos de carne a los ya existentes.
- `public void addPatatas(int x)`. Añade `x` kilos de patatas a los ya existentes.
- `public int getComensales()`. Devuelve el número de clientes que puede atender el restaurante (este es el método del ejercicio anterior).
- `public double getCarne()`. Devuelve los kilos de carne que hay en el almacén.
- `public double getPatatas()`. Devuelve los kilos de patatas que hay en el almacén.

14. (Proveedor) Crear un clase llamada `Proveedor` con las siguientes propiedades:

- `CIF`
- `nombreEmpresa`
- `descripcion`
- `sector`
- `direccion`

- telefono
- poblacion
- codPostal
- correo

Crear para la clase `Proveedor` los métodos:

- Constructor que permite crear una instancia con los datos de un proveedor.
  - Métodos get (*getters*).
  - Métodos set (*setters*).
  - Método `verificaCorreo` que devuelve true si la dirección de correo contiene @ .
  - Método que muestre todos los datos del proveedor.
- Crear un método principal `main` ejecutable que:
- Cree una instancia del objeto `Proveedor` llamado `proveedor`.
  - Cambie el sector del `proveedor`.
  - Muestre el sector del `proveedor`.
  - Verifique si el correo es válido.
  - Muestre todos los datos del `proveedor`.

15. (Producto) Crear una clase llamada `Producto` con las siguientes propiedades:

- codProducto
- nombreProducto
- descripcion
- categoria
- peso
- precio
- stock

Crear para la clase `Producto` los siguiente métodos:

- `Producto` : Permite crear una instancia con los datos de un producto.
- Getters y Setters para todas las propiedades.
- `aumentaStock` : Permite aumentar el stock de unidades del producto. Se le pasa el dato de unidades que aumentamos.
- `disminuyeStock` : Permite disminuir el stock de unidades del producto. Se le pasa el dato de unidades que disminuimos.
- `ivaProducto` : Permite calcular el IVA aplicado al precio del producto. Se le pasa el dato del porcentaje de IVA.
- `mostrarDatos` : Muestra los datos del producto. (No tiene test)

Crear un método principal `main` ejecutable que:

- Crear dos instancias de la clase `Producto` llamadas `productoHardware` y `productoSoftware`.
- Mostrar los datos de los dos objetos `Producto` que hemos creado.
- Aumenta el stock de unidades del `productoHardware` en 12 unidades.
- Disminuir el stock de unidades del `productoSoftware` en 5 unidades.
- Calcula el IVA de los dos objetos `Producto` que hemos creado.
- Mostrar los datos de los dos objetos `Producto`, así como sus importes de IVA y los precios finales de cada una de las instancias.

16. (Cuenta) Crea una clase llamada `Cuenta` que tendrá los siguientes atributos: `titular` y `cantidad` (puede tener decimales).

Al crear una instancia del objeto `Cuenta`, el titular será obligatorio y la cantidad es opcional. Crea dos constructores que cumplan lo anterior; es decir debemos crear dos métodos constructores con el mismo nombre que será el nombre del objeto.

Crea sus métodos get, set y el método `mostrarDatos` que muestre los datos de la cuenta. Tendrá dos métodos especiales:

- `ingresar(double cantidad)` : se ingresa una cantidad a la cuenta, si la cantidad introducida es negativa, no se hará nada.
- `retirar(double cantidad)` : se retira una cantidad a la cuenta, si restando la cantidad actual a la que nos pasan es negativa, la cantidad de la cuenta pasa a ser 0 retirando el importe máximo en función de la cantidad disponible en el objeto.

Crear una método principal `main` ejecutable:

- Crear una instancia del objeto `Cuenta` llamada `cuentaParticular1` con el nombre del titular.
- Crear una instancia del objeto `Cuenta` llamada `cuentaEmpresa1` con el nombre del titular y una cantidad inicial de dinero.
- Mostrar el titular de la instancia `cuentaParticular1`.
- Mostrar el saldo de la instancia `cuentaEmpresa1`.

- Ingresar 1000 € en la instancia `cuentaParticular1`.
- Retirar 500 € en la instancia `cuentaEmpresa1`.
- Mostrar los datos de las dos instancias del objeto `Cuenta`.

17. (Libro) Crea una clase llamada `Libro` que guarde la información de cada uno de los libros de una biblioteca. La clase debe guardar las siguientes propiedades:

- `título`
- `autor`
- `editorial`
- `número de ejemplares totales`
- `número de prestados`

La clase contendrá los siguientes métodos:

- Constructor por defecto.
- Constructor con parámetros.
- Métodos Setters/getters.
- Método `prestamo` que incremente el atributo correspondiente cada vez que se realice un préstamo del libro. No se podrán prestar libros de los que no queden ejemplares disponibles para prestar. Devuelve `true` si se ha podido realizar la operación y `false` en caso contrario.
- Método `devolucion` que decremente el atributo correspondiente cuando se produzca la devolución de un libro. No se podrán devolver libros que no se hayan prestado. Devuelve `true` si se ha podido realizar la operación y `false` en caso contrario.
- Método `perdido` que decremente el atributo número de ejemplares por perdida de ejemplar. No se podrán perder libros que no tengan ejemplares o no se hayan prestado. Devuelve `true` si se ha podido realizar la operación y `false` en caso contrario.
- Método `mostrarDatos` para mostrar los datos de los libros.

Crear un método principal `main` ejecutable:

- Crear una instancia del objeto libro `libroInformatica1` con los datos de un libro.
- Consultar el título de la instancia `libroInformatica1`.
- Cambiar la editorial de la instancia `libroInformatica1` por Anaya.
- Realiza el préstamo de la instancia `libroInformatica1`.
- Realiza otro préstamo de la instancia `libroInformatica1`.
- Muestra los préstamos de la instancia `libroInformatica1`.
- Realiza la devolución de la instancia `libroInformatica1`.
- Muestra los préstamos de la instancia `libroInformatica1`.
- Gestiona la pérdida de un ejemplar de la instancia `libroInformatica1`.
- Muestra los ejemplares de la instancia `libroInformatica1`.
- Muestra todos los datos de la instancia `libroInformatica1`.

18. (Hospital) Crear una clase llamada `Hospital` con las siguientes propiedades y métodos:

- Propiedades:
  - `codHospital`
  - `nombreHospital`
  - `direccion`
  - `telefono`
  - `poblacion`
  - `codPostal`
  - `habitacionesTotales`
  - `habitacionesOcupadas`
- Métodos:
  - `Hospital`: Permite crear una instancia con los datos de un hospital.
  - Métodos get.
  - Métodos set.
  - Método `ingreso` que incremente las habitaciones ocupadas. No puede realizarse el ingreso si las habitaciones ocupadas son iguales a las habitaciones totales del hospital. Devuelve `true` si se ha podido realizar el ingreso.

- Método `alta` que decrementa las habitaciones ocupadas. No puede realizarse el alta las habitaciones ocupadas son 0. Devuelve `true` si se ha podido realizar el alta.
- Método que muestre todos los datos del hospital.
- Crear un método principal `main` ejecutable que:
- Cree una instancia de la clase `Hospital` llamada `hospitalRibera`.
- Cambie el número de habitaciones de la instancia `hospitalRibera`.
- Muestre el número de habitaciones de la instancia `hospitalRibera`.
- Realiza un ingreso de la instancia `hospitalRibera`.
- Muestra las habitaciones ocupadas de la instancia `hospitalRibera`.
- Realiza un alta de la instancia `hospitalRibera`.
- Muestra las habitaciones ocupadas de la instancia `hospitalRibera`.
- Muestre todos los datos de la instancia `hospitalRibera`.

19. (Medico) Crear un clase llamada `Medico` con las siguientes propiedades y métodos:

- Propiedades:
  - `codMedico`
  - `nombre`
  - `apellidos`
  - `dni`
  - `direccion`
  - `telefono`
  - `poblacion`
  - `codPostal`
  - `fechaNacimiento`
  - `especialidad`
  - `sueldo`
- Métodos:
  - `Medico`: Permite crear una instancia con los datos de un médico.
  - Métodos `get`. Recuperan datos de la instancia del objeto.
  - Métodos `set`. Asignan datos a la instancia del objeto.
  - `retencionMedico` : Permite calcular la retención aplicada al sueldo del médico. Se le pasa el dato del porcentaje de retención.
  - `mostrarDatos` : Muestra los datos del médico.
  - Crear un método principal `main` ejecutable que:
  - Crear dos instancias de la clase `Medico` llamados `medicoDigestivo` y `medicoTraumatologo`.
  - Cambia el sueldo del `medicoTraumatologo`.
  - Muestra el sueldo del `medicoTraumatologo`.
  - Cambia el dni del `medicoDigestivo`.
  - Muestra el dni del `medicoDigestivo`.
  - Calcula la retención de las dos instancias de la clase `Medico` que hemos creado.
  - Mostrar los datos de las dos instancias de la clase `Medico` que hemos creado, así como las retenciones y los sueldos finales de cada una.

## 7.2. Ejercicios

---

Estos ejercicios utilizan la interfaz gráfica a la que dedicaremos más tiempo hacia finales de curso. De momento con entender algunos conceptos muy básicos de como dibujar elementos gráficos en una ventana podemos intentar resolverlos usando los conceptos de objetos, clases, herencia, etcétera que hemos visto en teoría.

El primero está resuelto y comentado para que te ayude a resolver el resto por tu cuenta o con la ayuda del docente.

1. (LlenarConCirculo) Crear una pizarra cuadrada y dibujar en ella un círculo que la ocupe por completo.

```

1 //importaciones necesarias para los ejercicios, no necesitas más.
2 import javax.swing.JFrame;
3 import javax.swing.JPanel;
4 import java.awt.Color;
5 import java.awt.Graphics;
6
7 /*
8 Necesitamos que nuestra clase LlenarConCirculo herede de JPanel para poder
9 pintar en su interior.
10 */
11 public class LlenarConCirculo extends JPanel {
12
13 @Override
14 public void paint(Graphics g) {
15 //Fijamos el color que tendrá la figura
16 g.setColor(Color.RED);
17
18 /* Dibujamos un ovalo relleno fijando las 4 esquinas que lo delimitan:
19 - x1, y1, x2, y2
20 En nuestro caso además hacemos uso de la función reflexiva
21 this.getWidth() y this.getHeight() para conocer la anchura y altura
22 (respectivamente) de nuestra ventana.
23 */
24 g.fillOval(0, 0, this.getWidth(), this.getHeight());
25
26 Otras funciones disponibles para dibujar son:
27 - fill3DRect(int x, int y, int width, int height, boolean raised)
28 Paints a 3-D highlighted rectangle filled with the current color.
29 - fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)
30 Fills a circular or elliptical arc covering the specified rectangle.
31 - fillOval(int x, int y, int width, int height)
32 Fills an oval bounded by the specified rectangle with the current color.
33 - fillPolygon(int[] xPoints, int[] yPoints, int nPoints)
34 Fills a closed polygon defined by arrays of x and y coordinates.
35 - fillPolygon(Polygon p)
36 Fills the polygon defined by the specified Polygon object with the graphics
37 context's current color.
38 - fillRect(int x, int y, int width, int height)
39 Fills the specified rectangle.
40 - fillRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)
41 Fills the specified rounded corner rectangle with the current color.
42 - fill3DRect(int x, int y, int width, int height, boolean raised)
43 Paints a 3-D highlighted rectangle filled with the current color.
44 - fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)
45 Fills a circular or elliptical arc covering the specified rectangle.
46 - fillOval(int x, int y, int width, int height)
47 Fills an oval bounded by the specified rectangle with the current color.
48 - fillPolygon(int[] xPoints, int[] yPoints, int nPoints)
49 Fills a closed polygon defined by arrays of x and y coordinates.
50 - fillPolygon(Polygon p)
51 Fills the polygon defined by the specified Polygon object with the graphics
52 context's current color.
53 - fillRect(int x, int y, int width, int height)
54 Fills the specified rectangle.
55 - fillRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)
56 Fills the specified rounded corner rectangle with the current color.
57 */
58 }
59
60 public static void main(String[] args) {
61 //Creamos una nueva ventana
62 JFrame MainFrame = new JFrame();
63 //Fijamos su tamaño en 300px de ancho por 300px de alto
64 MainFrame.setSize(300, 300);
65 //Creamos el objeto que vamos a dibujar con el método paint()
66 LlenarConCirculo circlePanel = new LlenarConCirculo();
67 //Añadimos el objeto recién creado a la ventana
68 MainFrame.add(circlePanel);
69 //Hacemos visible la ventana (con el dibujo)
70 MainFrame.setVisible(true);
71 }
72 }
```

Este es el esquema básico que necesitas para resolver todos los ejercicios planteados:

```

1 //importaciones necesarias para los ejercicios, no necesitas más.
2 import javax.swing.JFrame;
3 import javax.swing.JPanel;
4 import java.awt.Color;
5 import java.awt.Graphics;
6
7 /*
8 Necesitamos que nuestra clase herede de JPanel para poder
9 pintar en su interior.
10 */
11 public class TuClaseEjercicio extends JPanel {
12
13 @Override
14 public void paint(Graphics g) {
15 // INSERTA TU CÓDIGO AQUÍ!!! <-->
16 //Fijamos el color que tendrá la figura
17 //Dibuja la/s figura/s que te pide el ejercicio
18 }
19
20 public static void main(String[] args) {
21 //Creamos una nueva ventana
22 JFrame MainFrame = new JFrame();
23 //Fijamos su tamaño en 300px de ancho por 300px de alto
24 MainFrame.setSize(300, 300);
25 //Creamos el objeto que vamos a dibujar con el método paint()
26 LlenarConCirculo tuDibujo = new LlenarConCirculo();
27 //Añadimos el objeto recién creado a la ventana
28 MainFrame.add(tuDibujo);
29 //Hacemos visible la ventana (con el dibujo)
30 MainFrame.setVisible(true);
31 }
32 }
```

2. (LlenarConRectangulo) Crear una pizarra de tamaño aleatorio y dibujar en ella un rectángulo que la ocupe por completo.
3. (MitadYMitad) Crear una pizarra de tamaño aleatorio y dibujar un rectángulo ROJO que ocupe la mitad izquierda y uno VERDE que ocupe la mitad derecha.
4. (Dos partes) Crear una pizarra de tamaño aleatorio y dibujar un rectángulo ROJO que ocupe la parte superior (25% de la altura) y uno VERDE que ocupe la parte inferior (75% restante).
5. (CentrarFiguras) Crear una pizarra de tamaño aleatorio. Dibujar en el centro un cuadrado de lado 100 y un círculo de radio 25.
6. (RadioAleatorioCentrado) Crear una pizarra de tamaño aleatorio. Dibujar en centro de la pizarra un círculo de radio aleatorio (entre 50 y 200 pixels de radio)
7. (RadioAleatorio) Crear una pizarra de tamaño aleatorio. Dibujar en la esquina superior izquierda un círculo de radio aleatorio (entre 50 y 200)

 7 de noviembre de 2025

## 8. 3.3 Talleres

### 8.1. Taller UD02\_01: GitHub Classroom

#### 8.1.1. Requisitos previos

Necesitamos:

- Una cuenta de GitHub
- Tener el IDE IntelliJ instalado en nuestro ordenador

#### 8.1.2. Unirnos a GitHub Classroom

Aceptamos el *Assignment* (la tarea/ejercicio) a partir del link del profesor, en este caso: <https://classroom.github.com/a/LTEIJf5H>



1DAM\_Programacion\_25\_26

## Accept the assignment — UD02

Once you accept this assignment, you will be granted access to the `ud02-martinezpenya` repository in the [IES-Eduardo-Primo-Marques](#) organization on GitHub.

[Accept this assignment](#)



Nos mostrará la siguiente pantalla:



The header of the GitHub Classroom interface. It features the GitHub Classroom logo on the left, followed by a series of icons: GitHub Education, a message icon, a gear icon, a question mark icon, a user profile icon, and a refresh/circular arrow icon.



## You're ready to go!

You accepted the assignment, **UD02**.

Your assignment repository has been created:

 <https://github.com/IES-Eduardo-Primo-Marques/ud02-martinezpenya>

We've configured the repository associated with this assignment.



### Join the GitHub Student Developer Pack

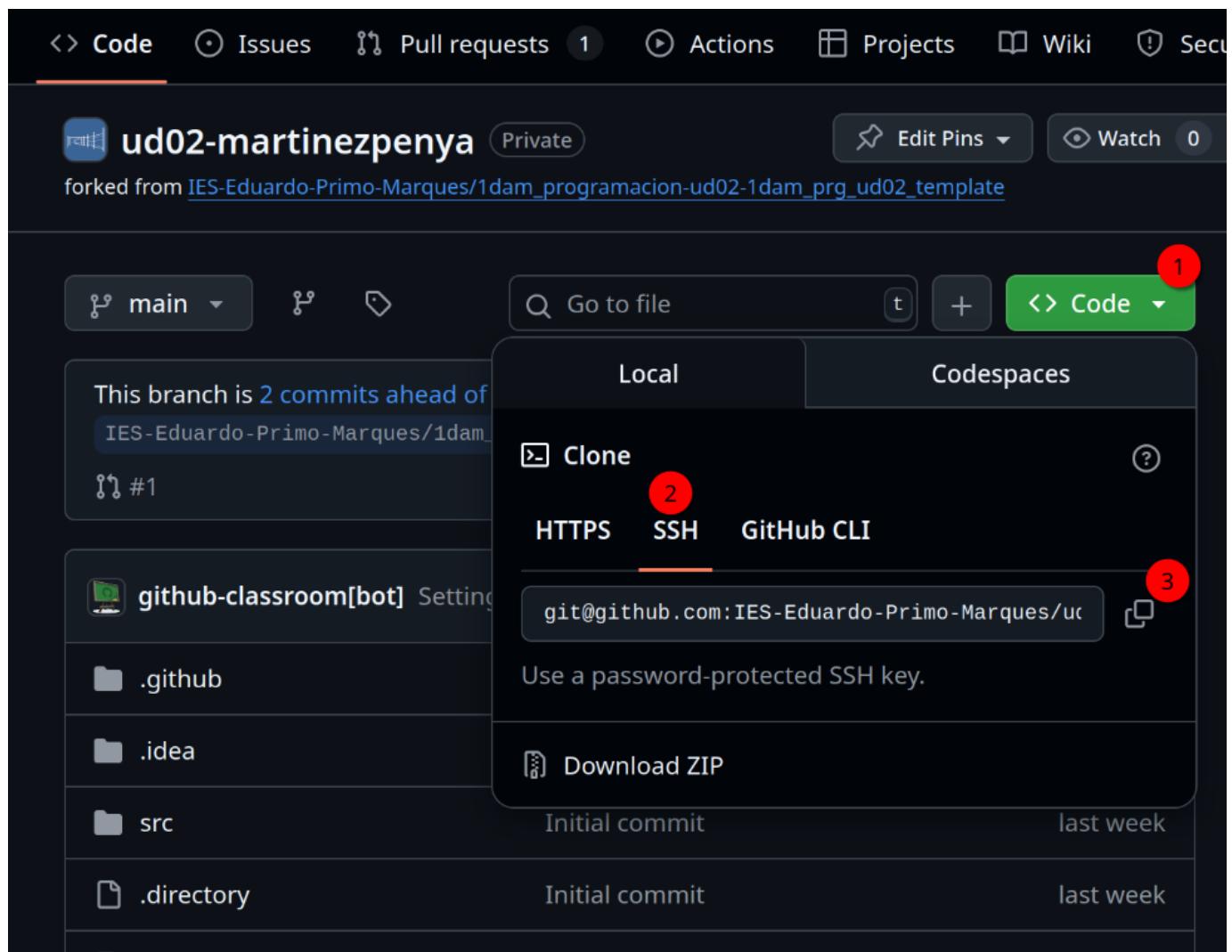
Verified students receive free GitHub Pro plus thousands of dollars worth of the best real-world tools and training from GitHub Education partners — for free. For more information, visit [GitHub Student Developer Pack](#).

[Apply](#)

Note: You may receive an email invitation to join [IES-Eduardo-Primo-Marques](#) on your behalf.  
No further action is necessary.

Abrimos el enlace que aparece con fondo azul: <https://github.com/IES-Eduardo-Primo-Marques/ud02-martinezpenya> (Vuestro enlace será diferente, este es el mio)

Esto nos lleva al repositorio en GitHub, y desde allí, copiamos la URL del repositorio, pero la de `ssh` en lugar de `https`:



#### 8.1.3. Preparar nuestra autenticación a GitHub mediante clave privada/pública (necesaria para usar ssh)

En agosto 2021 GitHub eliminó la autenticación por contraseña para operaciones Git en la línea de comandos. Desde el 13 de agosto de 2021, es OBLIGATORIO usar:

- Tokens de acceso personal (Personal Access Tokens - PAT) para HTTPS
- Claves SSH para conexiones SSH
- GitHub CLI con su propio sistema de autenticación

##### 8.1.3.1. PASO 1: GENERAR UNA NUEVA CLAVE SSH

1. Abre la consola o PowerShell

2. Genera la clave SSH:

```
1 ssh-keygen -t ed25519 -C "tu_email@ejemplo.com"
```

Reemplaza "tu\_email@ejemplo.com" con tu email de GitHub

Para sistemas más antiguos, usa: `ssh-keygen -t rsa -b 4096 -C "tu_email@ejemplo.com"`

3. Sigue las instrucciones:

```
1 Enter file in which to save the key (C:\Users\tunombre/.ssh/id_ed25519):
```

Presiona **Enter** para aceptar la ubicación por defecto

## 1. Establece una contraseña segura (opcional pero recomendado):

```
1 Enter passphrase (empty for no passphrase):
2 Enter same passphrase again:
```

## 2. Introduce una contraseña segura y confírmala.

### 8.1.3.2. PASO 2: LOCALIZAR Y COPIAR LA CLAVE PÚBLICA

1. Navega al directorio `.ssh`: Deberías ver `id_ed25519` (clave privada) y `id_ed25519.pub` (clave pública)
2. Copia la clave pública `id_ed25519.pub`:

Abre el archivo en el Bloc de notas y copia el contenido del fichero

### 8.1.3.3. PASO 3: INICIAR EL AGENTE Y AGREGAR LA CLAVE PRIVADA (OPCIONAL)

1. Abre PowerShell (esta vez necesitarás hacerlo como Administrador)

## 2. Verifica si el servicio SSH está instalado:

```
1 Get-WindowsCapability -Online | Where-Object Name -like 'OpenSSH*'
```

## 1. Si falta algún componente, instálalo:

```
1 #Instalar cliente SSH
2 Add-WindowsCapability -Online -Name OpenSSH.Client~~~~0.0.1.0
3 #Instalar servidor SSH
4 Add-WindowsCapability -Online -Name OpenSSH.Server~~~~0.0.1.0
```

## 1. Inicia el agente SSH:

```
1 # Iniciar servicio
2 Start-Service ssh-agent
3
4 # Configurar para inicio automático
5 Set-Service -Name ssh-agent -StartupType Automatic
```

## 1. Agrega tu clave SSH:

```
1 ssh-add $env:USERPROFILE\.ssh\id_ed25519
```

### 8.1.3.4. PASO 4: CONFIGURAR LA CLAVE SSH EN GITHUB

1. **Inicia sesión en tu cuenta de GitHub** - Ve a [github.com](https://github.com) y accede a tu cuenta
2. **Accede a la configuración de SSH**: - Haz clic en tu **foto de perfil** (esquina superior derecha) - Selecciona **Settings** - En el menú lateral, haz clic en **SSH and GPG keys**
3. **Agrega una nueva clave SSH**: - Haz clic en el botón **New SSH key** o **Add SSH key**
4. **Completa los campos**: - **Title**: Un nombre descriptivo (ej: "Mi PC Windows") - **Key type**: Dejar como "Authentication Key" - **Key**: Pega la clave pública que copiaste anteriormente
5. **Guarda la clave**: - Haz clic en **Add SSH key** - Confirma tu contraseña de GitHub si es necesario

### 8.1.3.5. PASO 5: VERIFICAR LA CONEXIÓN SSH

## 1. Prueba la conexión en la terminal o PowerShell:

Si ejecutas este comando:

```
1 ssh -T git@github.com
```

Deberías ver algo parecido a esto (después de contestar `yes` a la pregunta, fíjate que aparece tu nombre de GitHub, en mi caso `martinezpenya`):

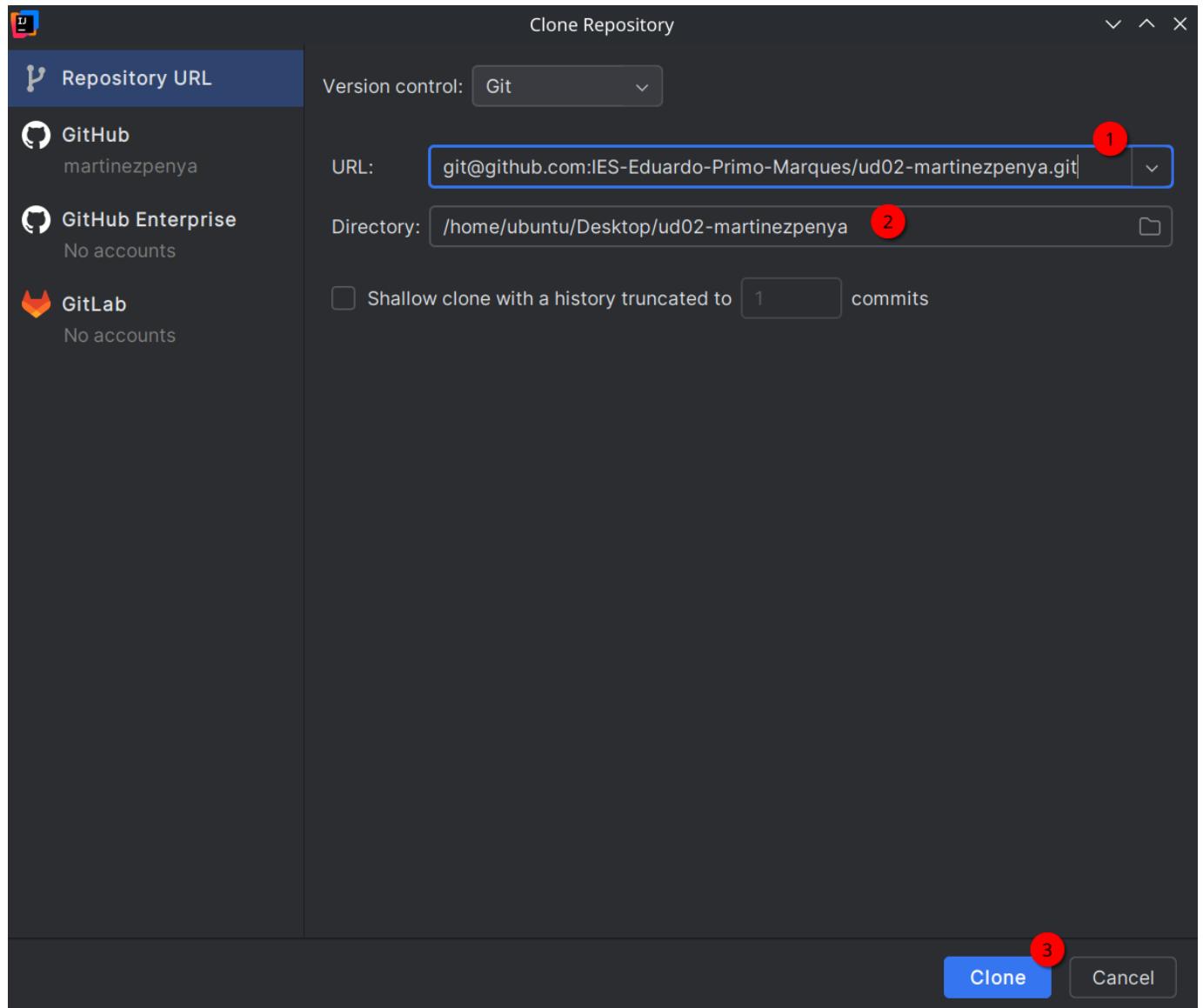
```

1 The authenticity of host 'github.com (140.82.121.3)' can't be established.
2 ED25519 key fingerprint is SHA256:+DiY3wvvV6TuJJhbpZisF/zLDA0zPMSvHdkr4UvC0qU.
3 This key is not known by any other names.
4 Are you sure you want to continue connecting (yes/no/[fingerprint])?yes
5 Warning: Permanently added 'github.com' (ED25519) to the list of known hosts.
6 Hi martinezpenya! You've successfully authenticated, but GitHub does not provide shell access.

```

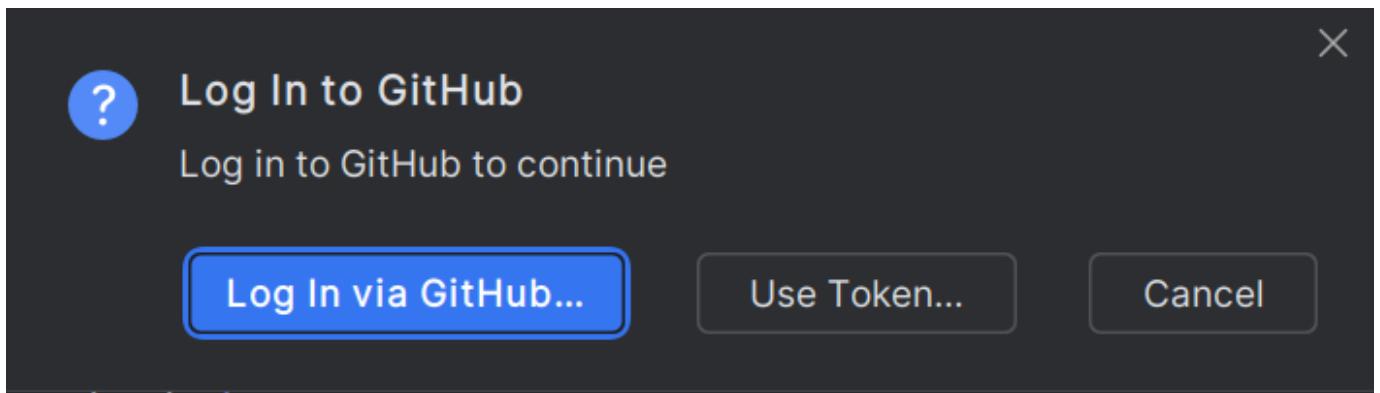
#### 8.1.4. Crear el proyecto en IntelliJ

Ahora abriremos IntelliJ y crearemos un nuevo proyecto (New Project from Version Control, o Project/Clone Repository) a partir de la url que hemos copiado en el punto anterior:

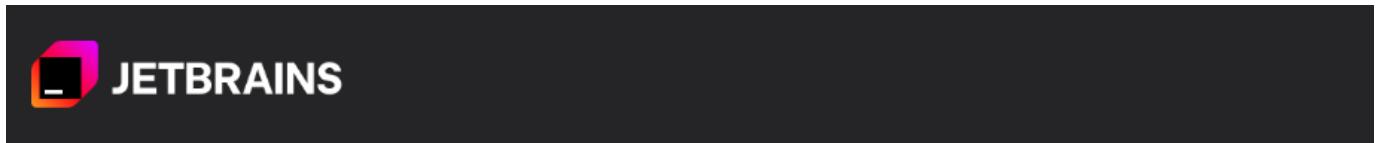


1. Pegamos la URL del paso anterior
2. Elegimos la ubicación de nuestro PC donde guardaremos el proyecto
3. Pulsamos el botón clonar

A continuación nos pedirá hacer Login con nuestra cuenta de GitHub:



Y en nuestro navegador debemos Autorizar la vinculación de GitHub en el IDE de JetBrains:



Y cuando todo esté correcto aparecerá:



Y ahora en nuestro IDE IntelliJ tenemos:



Ahora en la carpeta `src` debemos buscar el enunciado del ejercicio, en este caso `src/main/java/es/martinezpenya/UD02/_01_Temperatura.java`

```

1 package es.martinezpenya.UD02;
2
3 import java.util.Scanner;
4
5 /**
6 *
7 * @author David Martínez (https://www.martinezpenya.es | https://martinezpenya.es/1DAMProgramacion/)
8 */
9
10 /*
11 ENUNCIADO (Puntos: 2)
12 (Temperatura) Crear una clase llamada Temperatura con dos métodos:
13
14 - `celsiusToFahrenheit`. Convierte grados *Celsius* a *Farenheit*.
15 $$
16 F=(1,8*C)+32
17 $$
18
19 - `fahrenheitToCelsius`. Convierte grados *Farenheit* a *Celsius*.
20 $$
21 C=\frac{F-32}{1,8}
22 $$
23
24 */
25
26 public class _01_Temperatura {
27 /* TODO: Tu solución aquí */
28 }

```

Debemos leer el enunciado y escribir nuestro código donde pone:

```

1 ...
2 /* TODO: Tu solución aquí */
3 ...

```

Una vez comprobamos que el código funciona correctamente, vamos a comprobar si pasará los test impuestos por el profesor.

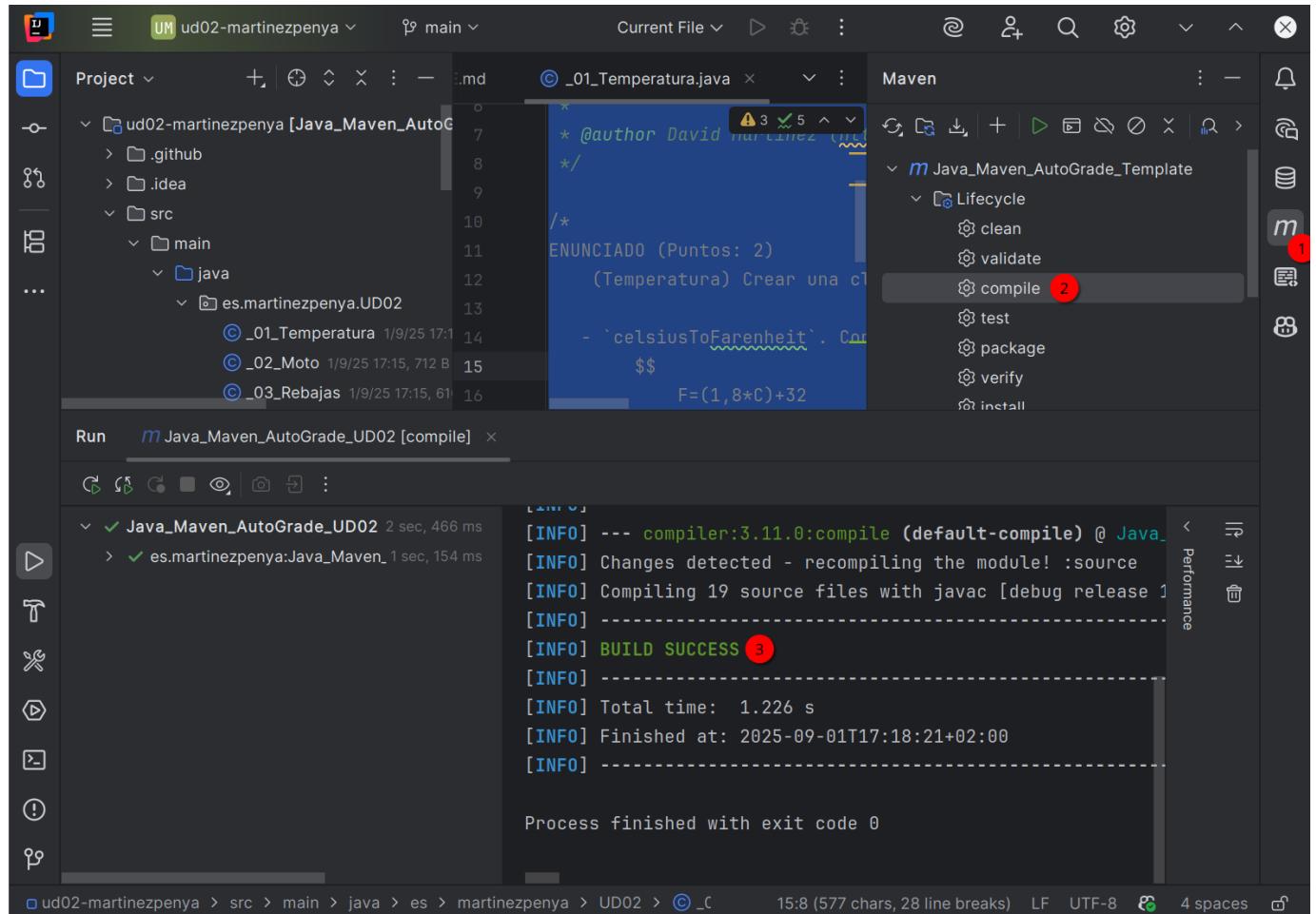
### ⚠️ Importante

En las siguientes capturas se asume que hemos resuelto correctamente el primer ejercicio, por tanto pasamos 2 de los 69 tests que contiene la UD02.

### 8.1.5. Comprobar los tests (Maven)

Debemos realizar dos acciones:

1. Primero compilar el proyecto completo con Maven:



The screenshot shows the IntelliJ IDEA interface with the Maven tool window open. In the Maven window, the 'Lifecycle' dropdown is expanded, and the 'compile' goal is selected, highlighted with a red circle. The output window below shows the Maven build logs, which end with '[INFO] BUILD SUCCESS' highlighted with a red circle. The status bar at the bottom indicates a total time of 1.226 s and a finished date of 2025-09-01T17:18:21+02:00.

```
[INFO] --- compiler:3.11.0:compile (default-compile) @ Java_Maven_AutoGrade_UD02
[INFO] Changes detected - recompiling the module! :source
[INFO] Compiling 19 source files with javac [debug release 1]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 1.226 s
[INFO] Finished at: 2025-09-01T17:18:21+02:00
[INFO] -----
```

Una vez comprobado que aparece **BUIL SUCCESS** (seguimos).

1. Lanzar la tarea test y comprobar que sea correcta:

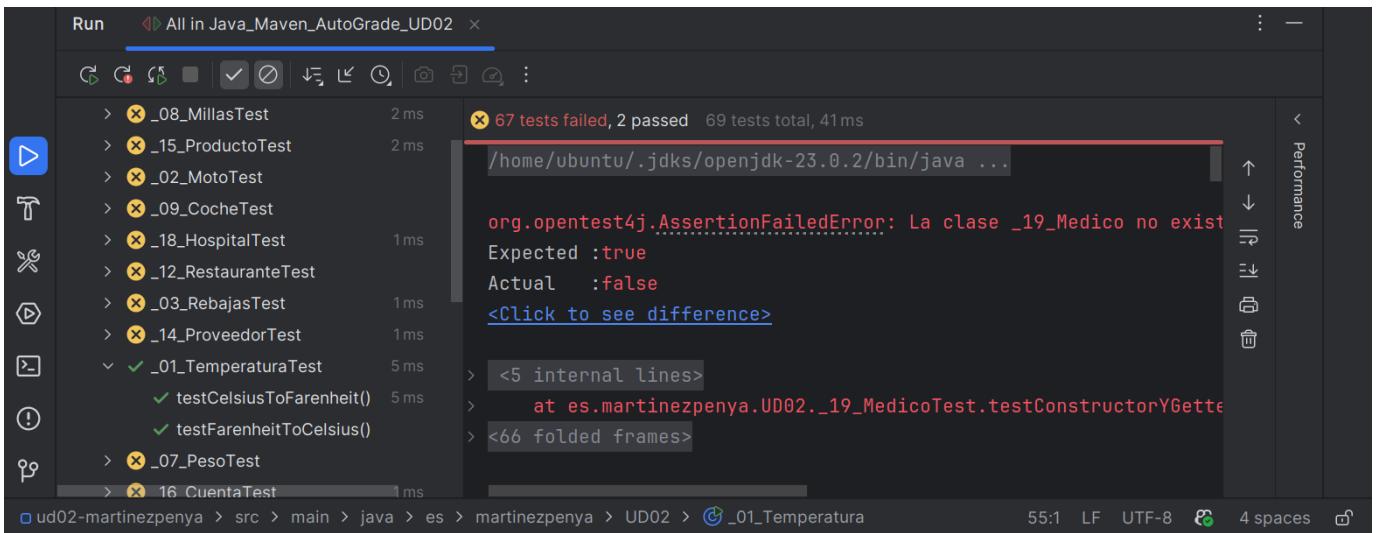
The screenshot shows the IntelliJ IDEA interface with a Java Maven project named 'ud02-martinezpenya'. The central editor displays the file '\_01\_Temperatura.java'. The Maven tool window on the right shows the 'Lifecycle' section with 'test' selected. The bottom pane shows the terminal output of the Maven test command, which failed with 69 errors. A green box highlights the error message: '[ERROR] Tests run: 69, Failures: 69, Errors: 0, Skipped: 0'.

O mejor todavía, podemos ejecutar todos los test con información más visual:

1. Buscamos la carpeta `test` en el proyecto
2. Pulsamos el botón derecho del ratón sobre la carpeta `java`
3. Elegimos la opción "Run 'All Tests'"



Y deberíamos ver un apartado similar en la parte inferior izquierda de nuestro IDE:

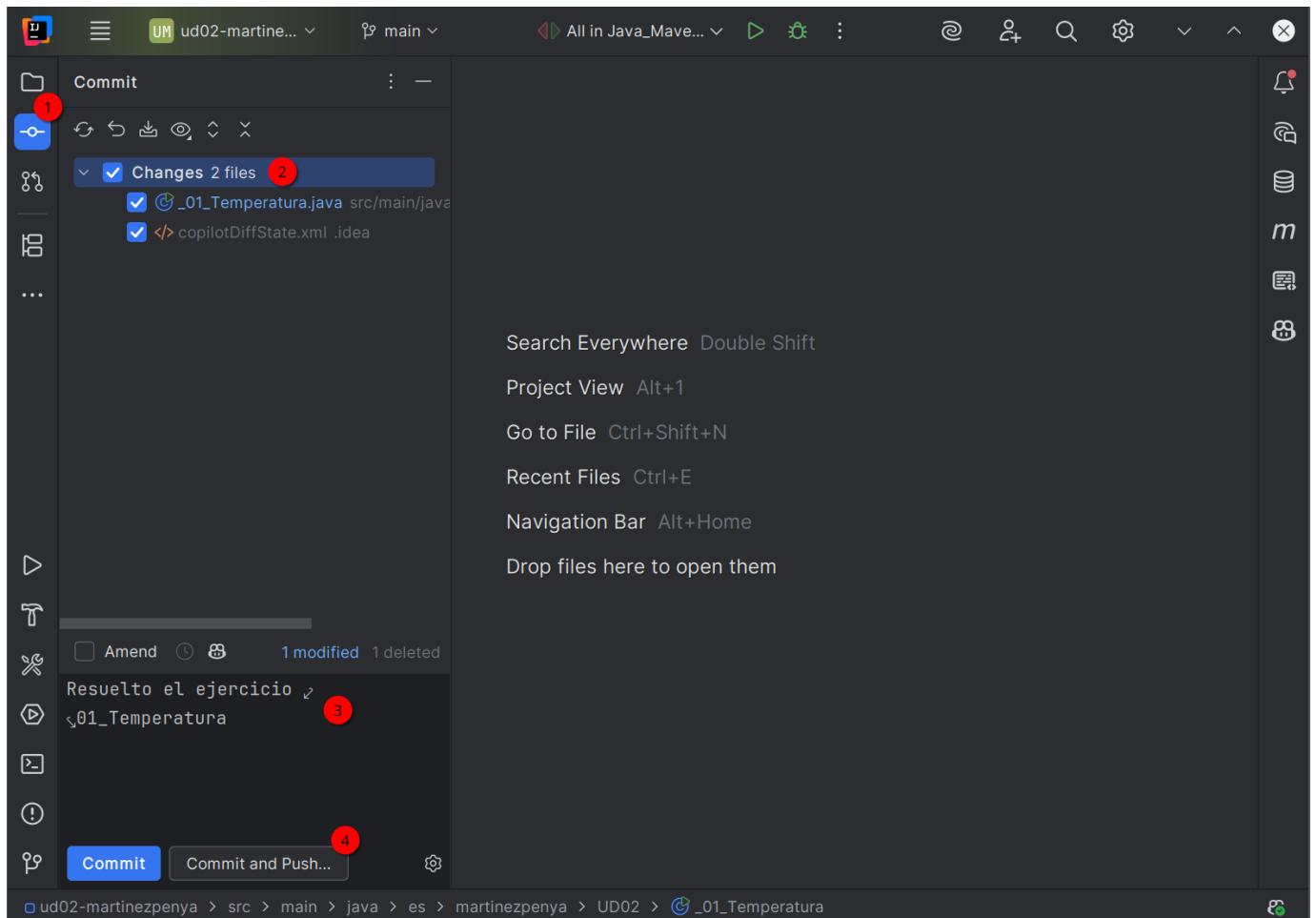


Debes tener en cuenta que:

1. Los tests no se ordenan alfabéticamente.
2. Si aparece un circulo amarillo con una X dentro, el test no pasa (aunque alguno de los tests internos funcionen).
3. Si el tick aparece en verde es que el test ha pasado.
4. Solo se conceden puntos en GitHub Classroom si todos los test de la clase funcionan. En este caso solo la clase `_01_Temperatura` nos otorgará puntos.

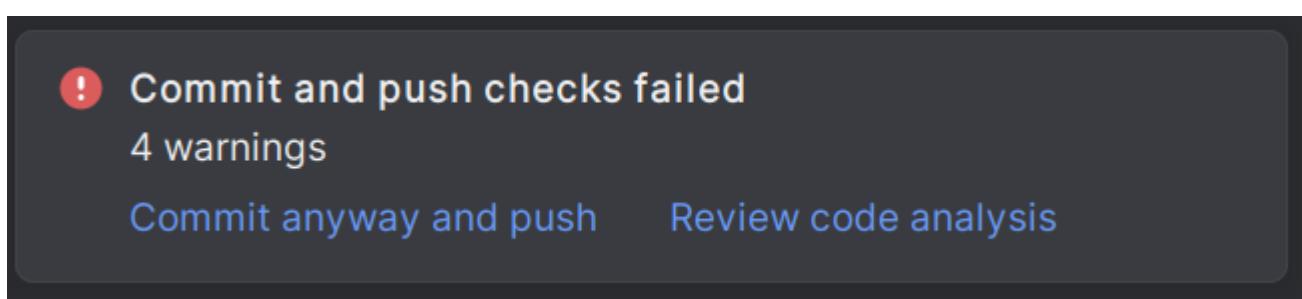
### 8.1.6. Subir nuestra solución a GitHub Classroom

Una vez comprobado que pasamos los tests lo que queda es subir nuestra solución a GitHub Classroom para que sea evaluada y valorada por el docente (Podemos repetir el proceso tantas veces como queramos, así que lo podemos ir haciendo según vayamos realizando los ejercicios).



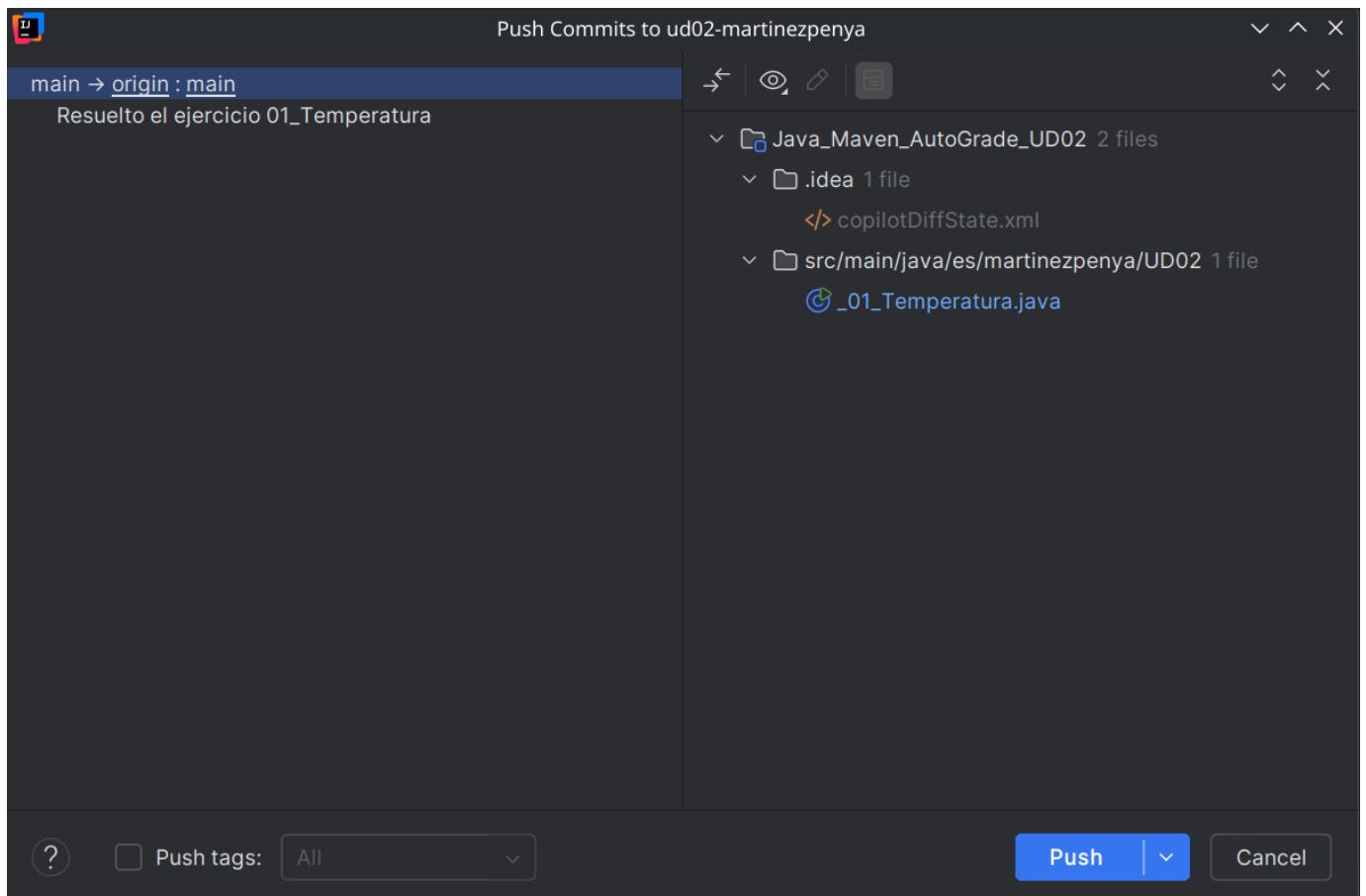
1. Pasamos al apartado de control de versiones
2. Elegimos los archivos que hemos cambiado y que queremos subir al repositorio para ser evaluados
3. Indicamos el mensaje del commit (recomiendo que los puedas identificar de alguna manera, indicando lo que has cambiado, un número de versión, etc.)
4. Finalmente pulsamos el botón Commit adn Push

Si aparece este error:



Podemos elegir la opción `Commit anyway and push` si hemos pasado alguno de los test correctamente, o bien `Review code analysis` para ver porque IntelliJ ha detectado algún problema.

Por último aparece esta pantalla, y pulsamos directamente el botón `Push`:



También podemos ver que test hemos pasado satisfactoriamente en nuestro repositorio de Github:

The screenshot shows the GitHub Classroom Actions page for a repository. The main header includes links for Code, Issues, Pull requests, Actions (with 1 update), Projects, Wiki, Security, Insights, and Settings. Below the header, a breadcrumb navigation shows Autograding Tests and the specific job title: Resuelto el ejercicio 01\_Temperatura #4. A "Re-run jobs" button and a three-dot menu are also present.

**Summary**

Triggered via push 4 minutes ago by **martinezpenya** pushed to **main**. Status: **Failure**. Total duration: **2m 21s**. Artifacts: **1**.

**Jobs**

- Preparación y ejecución de t... (Failed)
- Informe de Autograding (Passed)

**Run details**

**Usage**

**Workflow file**

**classroom.yml** on: push

**Preparación y ejecución de tests summary**

|                        | Tests  | Passed <input checked="" type="checkbox"/> | Skipped   | Failed <input checked="" type="checkbox"/> |
|------------------------|--------|--------------------------------------------|-----------|--------------------------------------------|
| Informe de Autograding | 69 ran | 2 passed                                   | 0 skipped | 67 failed                                  |

**Informe de Autograding**

| Test                                        | Result                                      |
|---------------------------------------------|---------------------------------------------|
| _01_TemperaturaTest.testCelsiusToFarenheit  | <input checked="" type="checkbox"/> passed  |
| _01_TemperaturaTest.testFarenheitToCelsius  | <input checked="" type="checkbox"/> passed  |
| <b>es.martinezpenya.UD02_01_MotoTest</b>    |                                             |
| _02_MotoTest.testAcelera                    | <input checked="" type="checkbox"/> failure |
| _02_MotoTest.testGetVelocidad               | <input checked="" type="checkbox"/> failure |
| _02_MotoTest.testFrena                      | <input checked="" type="checkbox"/> failure |
| <b>es.martinezpenya.UD02_02_BebidasTest</b> |                                             |

Por último, cuando el docente evalúe nuestra solución y si todo ha ido bien acabaremos viendo en GitHub Classroom que nuestra solución ha pasado los tests y nos ha asignado una puntuación:

Primero aparecerá como enviado (submitted):

A detailed view of a student submission card. It shows a profile picture of David Martínez, his name, and the word "Submitted". Below this, it shows his GitHub handle (@martinezpenya), the time of the latest commit (2 minutes ago), and the number of commits (1 commit).

Y cuando esté evaluado tendrá puntuación, en este caso 1 punto de un total de 1:

**David Martínez**

Submitted

@martinezpenya

Latest commit 5 minutes ago

-o- 1 commit

2/55

**8.1.7. OJO: No está permitido modificar archivos del proyecto que no estén en la carpeta src/java**

Aunque aparentemente todo te funcionará el docente será notificado e invalidará los resultados de los test:



Submitted

Protected file(s) modified

Latest commit last week

-o- 2 commits

7/55

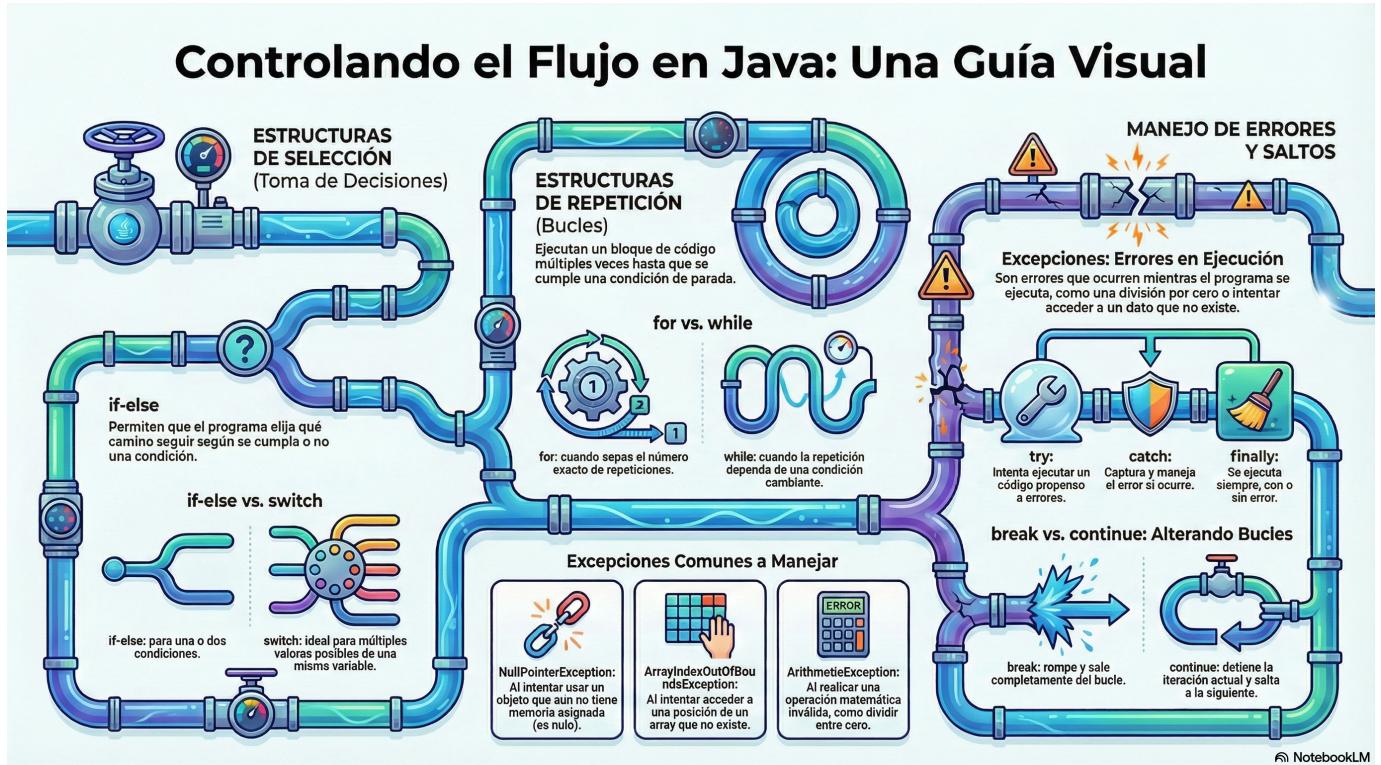
**8.1.8. Tarea**

Debes enviar tus soluciones a GitHub Classroom y superar al menos la mitad de los tests, cuantos más tests superados, mejor nota tendrás en la tarea.

⌚18 de octubre de 2025

## 4. UD03

### 9. 4.1 Estructuras de control y Excepciones



#### 9.1. Introducción

En unidades anteriores has podido aprender cuestiones básicas sobre el lenguaje JAVA: definición de variables, tipos de datos, asignación de valores, uso de literales, diferentes operadores que se pueden aplicar, conversiones de tipos, inserción de comentarios, etc. Posteriormente, nos sumergimos de lleno en el mundo de los objetos. Primero hemos conocido su filosofía, para más tarde ir recorriendo los conceptos y técnicas más importantes relacionadas con ellos: Propiedades, métodos, clases, declaración y uso de objetos, librerías, etc.

Vale, parece ser que tenemos los elementos suficientes para comenzar a generar programas escritos en JAVA, ¿Seguro?

Como habrás deducido, con lo que sabemos hasta ahora no es suficiente. Existen múltiples situaciones que nuestros programas deben representar y que requieren tomar ciertas decisiones, ofrecer diferentes alternativas o llevar a cabo determinadas operaciones repetitivamente para conseguir sus objetivos.

Si has programado alguna vez o tienes ciertos conocimientos básicos sobre lenguajes de programación, sabes que la gran mayoría de lenguajes poseen estructuras que permiten a los programadores controlar el flujo de la información de sus programas. Esto realmente es una ventaja para la persona que está aprendiendo un nuevo lenguaje, o tiene previsto aprender más de uno, ya que estas estructuras suelen ser comunes a todos (con algunos cambios de sintaxis o conjunto de reglas que definen las secuencias correctas de los elementos de un lenguaje de programación.). Es decir, si conocías sentencias de control de flujo en otros lenguajes, lo que vamos a ver a lo largo de esta unidad te va a sonar bastante.

Para alguien que no ha programado nunca, un ejemplo sencillo le va a permitir entender qué es eso de las sentencias de control de flujo. Piensa en un fontanero (programador), principalmente trabaja con agua (datos) y se encarga de hacer que ésta fluya por donde él quiere (programa) a través de un conjunto de tuberías, codos, latiguillos, llaves de paso, etc. (sentencias de control de flujo).

Pues esas estructuras de control de flujo son las que estudiaremos, conoceremos su estructura, funcionamiento, cómo utilizarlas y dónde. A través de ellas, al construir nuestros programas podremos hacer que los datos (agua) fluyan por los caminos adecuados para representar la realidad del problema y obtener un resultado adecuado.

Los tipos de estructuras de programación que se emplean para el control del flujo de los datos son los siguientes:

- **Secuencia:** compuestas por \(\{0\}\), \(\{1\}\) o \(\{N\}\) sentencias que se ejecutan en el orden en que han sido escritas. Es la estructura más sencilla y sobre la que se construirán el resto de estructuras.
- **Selección:** es un tipo de sentencia especial de decisión y de un conjunto de secuencias de instrucciones asociadas a ella. Según la evaluación de la sentencia de decisión se generará un resultado (que suele ser verdadero o falso) y en función de éste, se ejecutarán una secuencia de instrucciones u otra. Las estructuras de selección podrán ser simples, compuestas y múltiples.
- **Iteración:** es un tipo de sentencia especial de decisión y una secuencia de instrucciones que pueden ser repetidas según el resultado de la evaluación de la sentencia de decisión. Es decir, la secuencia de instrucciones se ejecutará repetidamente si la sentencia de decisión arroja un valor correcto, en otro la estructura de repetición se detendrá.

Además de las sentencias típicas de control de flujo, en esta unidad haremos una revisión de las sentencias de salto, que aunque no son demasiado recomendables, es necesario conocerlas. Como nuestros programas podrán generar errores y situaciones especiales, echaremos un vistazo al manejo de excepciones en JAVA. Posteriormente, analizaremos la mejor manera de llevar a cabo las pruebas de nuestros programas y la depuración de los mismos. Y finalmente, aprenderemos a valorar y utilizar las herramientas de documentación de programas.

## 9.2. Sentencias y bloques

Este epígrafe lo utilizaremos para reafirmar cuestiones que son obvias y que en el transcurso de anteriores unidades se han dado por sabidas. Aunque, a veces, es conveniente recordar. Lo haremos como un conjunto de FAQ's:

- **¿Cómo se escribe un programa sencillo?** Si queremos que un programa sencillo realice instrucciones o sentencias para obtener un determinado resultado, es necesario colocar éstas una detrás de la otra, exactamente en el orden en que deben ejecutarse.
- **¿Podrían colocarse todas las sentencias una detrás de otra, separadas por puntos y comas en una misma línea?**, claro que sí, pero no es muy recomendable. Cada sentencia debe estar escrita en una línea, de esta manera tu código será mucho más legible y la localización de errores en tus programas será más sencilla y rápida. De hecho, cuando se utilizan herramientas de programación, los errores suelen asociarse a un número o números de línea.
- **¿Puede una misma sentencia ocupar varias líneas en el programa?**, sí. Existen sentencias que, por su tamaño, pueden generar varias líneas. Pero siempre finalizarán con un punto y coma.
- **¿En Java todas las sentencias se terminan con punto y coma?**, Efectivamente. Si detrás de una sentencia ha de venir otra, pondremos un punto y coma. Escribiendo la siguiente sentencia en una nueva línea. Pero en algunas ocasiones, sobre todo cuando utilizamos estructuras de control de flujo, detrás de la cabecera de una estructura de este tipo no debe colocarse punto y coma. No te preocupes, lo entenderás cuando analicemos cada una de ellas.
- **¿Qué es la sentencia nula en Java?** La sentencia nula es una línea que no contiene ninguna instrucción y en la que sólo existe un punto y coma. Como su nombre indica, esta sentencia no hace nada.
- **¿Qué es un bloque de sentencias?** Es un conjunto de sentencias que se encierra entre llaves y que se ejecutaría como si fuera una única orden. Sirve para agrupar sentencias y para clarificar el código. Los bloques de sentencias son utilizados en Java en la práctica totalidad de estructuras de control de flujo, clases, métodos, etc. La siguiente tabla muestra dos formas de construir un bloque de sentencias.

| Bloque de sentencias 1                     | Bloque de sentencias 2                                          |
|--------------------------------------------|-----------------------------------------------------------------|
| {sentencia1; sentencia2; ...; sentenciaN;} | {<br>sentencia1;<br>sentencia2;<br>...<br>sentenciaN;         } |

- **¿En un bloque de sentencias, éstas deben estar colocadas con un orden exacto?** En ciertos casos sí, aunque si al final de su ejecución se obtiene el mismo resultado, podrían ocupar diferentes posiciones en nuestro programa.

**DEBES CONOCER** Observa los tres archivos que te ofrecemos a continuación y compara su código fuente. Verás que los tres obtienen el mismo resultado, pero la organización de las sentencias que los componen es diferente entre ellos.

**Ejemplo 1:**

```

1 package organizacion_sentencias1;
2 /**
3 *
4 * Organización de sentencias secuencial
5 */
6 public class Organizacion_sentencias_1 {
7 public static void main(String[] args) {
8 System.out.println ("Organización secuencial de sentencias");
9 int dia=12;
10 System.out.println ("El dia es: " + dia);
11 int mes=11;
12 System.out.println ("El mes es: " + mes);
13 int anio=2011;
14 System.out.println ("El año es: " + anio);
15 }
16 }
```

En este primer archivo, las sentencias están colocadas en orden secuencial.

**Ejemplo 2:**

```

1 package organizacion_sentencias2;
2 /**
3 *
4 * Organización de sentencias con declaración previa de variables
5 */
6 public class Organizacion_sentencias_2 {
7 public static void main(String[] args) {
8 // Zona de declaración de variables
9 int dia=10;
10 int mes=11;
11 int anio=2011;
12 System.out.println ("Organización con declaración previa de variables");
13 System.out.println ("El día es: " + dia);
14 System.out.println ("El mes es: " + mes);
15 System.out.println ("El año es: " + anio);
16 }
17 }
```

En este segundo archivo, se declaran al principio las variables necesarias. En Java no es imprescindible hacerlo así, pero sí que antes de utilizar cualquier variable ésta debe estar previamente declarada. Aunque la declaración de dicha variable puede hacerse en cualquier lugar de nuestro programa.

**Ejemplo 3:**

```

1 package organizacion_sentencias3;
2 /**
3 *
4 * Organización de sentencias en zonas diferenciadas
5 * según las operaciones que se realicen en el código
6 */
7 public class Organizacion_sentencias_3 {
8 public static void main(String[] args) {
9 // Zona de declaración de variables
10 int dia;
11 int mes;
12 int anio;
13 String fecha;
14 //Zona de inicialización o entrada de datos
15 dia=10;
16 mes=11;
17 anio=2011;
18 fecha="";
19 //Zona de procesamiento
20 fecha=dia+"/"+mes+"/"+anio;
21 //Zona de salida
22 System.out.println ("Organización con zonas diferenciadas en el código");
23 System.out.println ("La fecha es: " + fecha);
24 }
25 }
```

En este tercer archivo, podrás apreciar que se ha organizado el código en las siguientes partes: declaración de variables, petición de datos de entrada, procesamiento de dichos datos y obtención de la salida. Este tipo de organización está más estandarizada y hace que nuestros programas ganen en legibilidad.

### Acción

Construyas de una forma o de otra tus programas, debes tener en cuenta siempre en Java las siguientes premisas: - **Declara** cada variable antes de utilizarla. - **Inicializa** con un valor cada variable la primera vez que la utilices. No es recomendable usar variables no inicializadas en nuestros programas, pueden provocar errores o resultados imprevistos.

## 9.3. Estructuras de selección

¿Cómo conseguimos que nuestros programas puedan tomar decisiones? Para comenzar, lo haremos a través de las estructuras de selección. Estas estructuras constan de una sentencia especial de decisión y de un conjunto de secuencias de instrucciones.

El funcionamiento es sencillo, la sentencia de decisión será evaluada y ésta devolverá un valor (verdadero o falso), en función del valor devuelto se ejecutará una secuencia de instrucciones u otra.

Por ejemplo, si el valor de una variable es mayor o igual que 5 se imprime por pantalla la palabra APROBADO y si es menor, se imprime SUSPENSO. Para este ejemplo, la comprobación del valor de la variable será la sentencia especial de decisión. La impresión de la palabra APROBADO será una secuencia de instrucciones y la impresión de la palabra SUSPENSO será otra. Cada secuencia estará asociada a cada uno de los resultados que puede arrojar la evaluación de la sentencia especial de decisión. Las estructuras de selección se dividen en:

1. Estructuras de selección simples o estructura if.
2. Estructuras de selección compuesta o estructura ifelse.
3. Estructuras de selección basadas en el operador condicional.
4. Estructuras de selección múltiples o estructura switch.

A continuación, detallaremos las características y funcionamiento de cada una de ellas. Es importante que a través de los ejemplos que vamos a ver, puedas determinar en qué circunstancias utilizar cada una de estas estructuras. Aunque un mismo problema puede ser resuelto con diferentes estructuras e incluso, con diferentes combinaciones de éstas.

### 9.3.1. Estructura if, if else, if else if

La estructura `if` es una estructura de selección o estructura condicional, en la que se evalúa una expresión lógica o sentencia de decisión y en función del resultado, se ejecuta una sentencia o un bloque de éstas. La estructura `if` puede presentarse de las siguientes formas:

#### Estructura if simple:

```
1 if (expresión-lógica)
2 sentencia1;
```

```
1 if (expresión-lógica){
2 sentencia1;
3 sentencia2;
4 ...
5 sentenciaN;
6 }
```

Si la evaluación de la expresión-lógica ofrece un resultado verdadero, se ejecuta la sentencia1 o bien el bloque de sentencias asociado. Si el resultado de dicha evaluación es falso, no se ejecutará ninguna instrucción asociada a la estructura condicional.

### Estructura `if` de doble alternativa.

```

1 if (expresión-lógica)
2 sentencia1;
3 else
4 sentencia2;
5 sentencia3;

```

```

1 if (expresión-lógica){
2 sentencia1;
3 ...
4 sentenciaN;
5 } else {
6 sentencia1;
7 ...
8 sentenciaN;
9 }

```

Si la evaluación de la expresión-lógica ofrece un resultado verdadero, se ejecutará la primera sentencia o el primer bloque de sentencias. Si, por el contrario, la evaluación de la expresión-lógica ofrece un resultado falso, no se ejecutará la primera sentencia o el primer bloque y sí se ejecutará la segunda sentencia o el segundo bloque.

#### Ejemplo

Haciendo una interpretación cercana al pseudocódigo tendríamos que si se cumple la condición (expresión lógica), se ejecutará un conjunto de instrucciones y si no se cumple, se ejecutará otro conjunto de instrucciones.

Hay que tener en cuenta que la cláusula `else` de la sentencia `if` no es obligatoria. En algunos casos no necesitaremos utilizarla, pero sí se recomienda cuando es necesario llevar a cabo alguna acción en el caso de que la expresión lógica no se cumpla.

En aquellos casos en los que no existe cláusula `else`, si la expresión lógica es falsa, simplemente se continuarán ejecutando las siguientes sentencias que aparezcan bajo la estructura condicional `if`.

Los condicionales `if` e `if-else` pueden anidarse, de tal forma que dentro de un bloque de sentencias puede incluirse otro `if` o `if-else`. El nivel de anidamiento queda a criterio del programador, pero si éste es demasiado profundo podría provocar problemas de eficiencia y legibilidad en el código. En otras ocasiones, un nivel de anidamiento excesivo puede denotar la necesidad de utilización de otras estructuras de selección más adecuadas.

Cuando se utiliza anidamiento de este tipo de estructuras, es necesario poner especial atención en saber a qué `if` está asociada una cláusula `else`. Normalmente, un `else` estará asociado con el `if` inmediatamente superior o más cercano que exista dentro del mismo bloque y que no se encuentre ya asociado a otro `else`.

### Estructura `if else if`.

Esta estructura es una alternativa a la anidación de sentencias `if else` funciona de modo que si se cumple una condición ejecuta unas sentencias y el caso contrario comprueba otra condición ejecutando unas sentencias si se cumple y así sucesivamente.

Veamos un ejemplo con `if` anidados:

```

1 if (condicion1) {
2 sentencias1;
3 } else {
4 if (condicion2) {
5 sentencias2;
6 } else {
7 if (condicion3) {
8 sentencias3;
9 } else {
10 sentencias4;
11 }
12 }
13 }

```

**El mismo ejemplo usando `if` `else if` quedaría de este modo:**

```

1 if (condicion1) {
2 sentencias1;
3 } else if (condicion2) {
4 sentencias2;
5 } else if (condicion3) {
6 sentencias3;
7 } else {
8 sentencias4;
9 }

```

### 9.3.2. Estructura `switch`

¿Qué podemos hacer cuando nuestro programa debe elegir entre más de dos alternativas?, una posible solución podría ser emplear estructuras `if` anidadas, aunque no siempre esta solución es la más eficiente. Cuando estamos ante estas situaciones podemos utilizar la estructura de selección múltiple `switch`. En la siguiente tabla se muestra tanto la sintaxis, como el funcionamiento de esta estructura.

**Sintaxis `switch`:**

```

1 switch (expresion) {
2 case valor1:
3 sentencia1_1;
4 sentencia1_2;
5 ...
6 break;
7 case valor2:
8 ...
9 case valorN:
10 sentenciaN_1;
11 sentenciaN_2;
12 ...
13 break;
14 default:
15 sentencias-default;
16 }

```

#### Condiciones:

- Donde `expresión` debe ser del tipo `char`, `byte`, `short` o `int`, y las constantes de cada `case` deben ser de este tipo o de un tipo compatible.
- La `expresión` debe ir entre paréntesis.
- Cada `case` llevará asociado un `valor` y se finalizará con dos puntos (`:`).
- El bloque de sentencias asociado a la cláusula `default` puede finalizar con una sentencia de ruptura `break` o no.

#### Funcionamiento:

- Las diferentes alternativas de esta estructura estarán precedidas de la cláusula `case` que se ejecutará cuando el valor asociado al `case` coincida con el valor obtenido al evaluar la expresión del `switch`.
- En las cláusulas `case`, no pueden indicarse expresiones condicionales, rangos de valores o listas de valores. (otros lenguajes de programación sí lo permiten). Habrá que asociar una cláusula `case` a cada uno de los valores que deban ser tenidos en cuenta.
- La cláusula `default` será utilizada para indicar un caso por defecto, las sentencias asociadas a la cláusula `default` se ejecutarán si ninguno de los valores indicados en las cláusulas `case` coincide con el resultado de la evaluación de la expresión de la estructura `switch`.
- La cláusula `default` puede no existir, y por tanto, si ningún `case` ha sido activado finalizaría el `switch`.
- Cada cláusula `case` puede llevar asociadas una o varias sentencias, sin necesidad de delimitar dichos bloques por medio de llaves.
- En el momento en el que el resultado de la evaluación de la expresión coincide con alguno de los valores asociados a las cláusulas `case`, se ejecutarán todas las instrucciones asociadas hasta la aparición de una sentencia `break` de ruptura. (la sentencia `break` se analizará en epígrafes posteriores)

##### 9.3.2.1. EXPRESIONES SWITCH MEJORADAS

En las [novedades de Java 12](#) se añadió la posibilidad de los `switch` fueran expresiones que retornan un valor en vez de sentencias y se evita el uso de la palabra reservada `break`.

## Sentencia Switch mejorada

```

1 int entero = 5;
2
3 String numericString = switch (entero) {
4 case 0 -> "cero";
5 case 1, 3, 5, 7, 9 -> "impar";
6 case 2, 4, 6, 8, 10 -> "par";
7 default -> "error";
8 };
9 System.out.println(numericString); //impar

```

En Java 13 en vez de únicamente el valor a retornar se permite crear bloques de sentencias para cada rama `case` y retornar el valor con la palabra reservada `yield`. En los bloques de sentencias puede haber algún cálculo más complejo que directamente retornar el valor deseado.

```

1 int entero2 = 4;
2
3 String numericString2 = switch (entero2) {
4 case 0 -> {
5 String value = calculaCero();
6 yield value;
7 }
8 case 1, 3, 5, 7, 9 -> {
9 String value = calculaImpar();
10 yield value;
11 }
12
13 case 2, 4, 6, 8, 10 -> {
14 String value = calculaPar();
15 yield value;
16 }
17
18 default -> {
19 String value = calculaDefecto();
20 yield value;
21 }
22 };
23 System.out.println(numericString); //calculaPar()

```

En resumen, se ha de comparar el valor de una expresión con un conjunto de constantes, si el valor de la expresión coincide con algún valor de dichas constantes, se ejecutarán los bloques de instrucciones asociados a cada una de ellas. Si no existiese coincidencia, se ejecutarían una serie de instrucciones por defecto.

## 9.4. Estructuras de repetición

Nuestros programas ya son capaces de controlar su ejecución teniendo en cuenta determinadas condiciones, pero aún hemos de aprender un conjunto de estructuras que nos permita repetir una secuencia de instrucciones determinada. La función de estas estructuras es repetir la ejecución de una serie de instrucciones teniendo en cuenta una condición.

A este tipo de estructuras se las denomina estructuras de repetición, estructuras repetitivas, bucles o estructuras iterativas. En Java existen cuatro clases de bucles:

- Bucle `for` (repite para)
- Bucle `for/in` (repite para cada), aka `for each`
- Bucle `while` (repite mientras)
- Bucle `do while` (repite hasta)

Los bucles `for` y `for/in` se consideran bucles controlados por contador. Por el contrario, los bucles `while` y `do...while` se consideran bucles controlados por sucesos.

La utilización de unos bucles u otros para solucionar un problema dependerá en gran medida de las siguientes preguntas:

- ¿Sabemos a priori cuántas veces necesitamos repetir un conjunto de instrucciones?
- ¿Sabemos si hemos de repetir un conjunto de instrucciones si una condición satisface un conjunto de valores?
- ¿Sabemos hasta cuándo debemos estar repitiendo un conjunto de instrucciones?
- ¿Sabemos si hemos de estar repitiendo un conjunto de instrucciones mientras se cumpla una condición?

Estas y otras preguntas tendrán su respuesta en cuanto analicemos cada una de estructuras repetitivas en detalle.

### Definición

Estudia cada tipo de estructura repetitiva, conoce su funcionamiento y podrás llegar a la conclusión de que algunos de estos bucles son equivalentes entre sí. Un mismo problema, podrá ser resuelto empleando diferentes tipos de bucles y obtener los mismos resultados.

#### 9.4.1. Estructura `for`

Hemos indicado anteriormente que el bucle `for` es un bucle controlado por contador. Este tipo de bucle tiene las siguientes características:

- Se ejecuta un número determinado de veces.
- Utiliza una variable contadora que controla las iteraciones del bucle.

En general, existen tres operaciones que se llevan a cabo en este tipo de bucles:

- Se inicializa la variable contadora.
- Se evalúa el valor de la variable contador, por medio de una comparación de su valor con el número de iteraciones especificado.
- Se modifica o actualiza el valor del contador a través de incrementos o decrementos de éste, en cada una de las iteraciones.

### Ejemplo

La inicialización de la variable contadora debe realizarse correctamente para garantizar que el bucle lleve a cabo, al menos, la primera repetición de su código interno. La condición de terminación del bucle debe variar en el interior del mismo, de no ser así, podemos caer en la creación de un bucle infinito. Cuestión que se debe evitar por todos los medios. Es necesario estudiar el número de veces que se repite el bucle, pues debe ajustarse al número de veces estipulado.

#### Sintaxis estructura `for` con una única sentencia:

```
1 for (inicialización; condición; iteración)
2 sentencia;
```

#### Sintaxis estructura `for` con un bloque de sentencias:

```
1 for (inicialización; condición; iteración) {
2 sentencia1;
3 sentencia2;
4 ...
5 sentenciaN;
6 }
```

Donde....:

- `inicialización` es una expresión en la que se inicializa una variable de control, que será la encargada de controlar el final del bucle.
- `condición` es una expresión que evaluará la variable de control. Mientras la condición sea falsa, el cuerpo del bucle estará repitiéndose. Cuando la condición se cumpla, terminará la ejecución del bucle.
- `iteración` indica la manera en la que la variable de control va cambiando en cada iteración del bucle. Podrá ser mediante incremento o decremento, y no solo de uno en uno.

#### 9.4.2. Estructura `for / in`

Junto a la estructura `for`, `for / in` también se considera un bucle controlado por contador. Este bucle es una mejora incorporada en la versión 5.0 de Java.

Este tipo de bucles permite realizar recorridos sobre arrays y colecciones de objetos. Los arrays son colecciones de variables que tienen el mismo tipo y se referencian por un nombre común. Así mismo, las colecciones de objetos son objetos que se dice son iterables, o que se puede iterar sobre ellos.

Este bucle es nombrado también como bucle `for` mejorado, o bucle `foreach`. En otros lenguajes de programación existen bucles muy parecidos a este.

#### La sintaxis `for` es la siguiente:

```
1 for (declaración: expresión) {
2 sentencia1;
3 ...
4 sentenciaN;
5 }
```

Donde....:

- `expresión` es un array o una colección de objetos.
- `declaración` es la declaración de una variable cuyo tipo sea compatible con `expresión`. Normalmente, será el tipo y el nombre de la variable a declarar.

El funcionamiento consiste en que para cada elemento de la expresión, guarda el elemento en la variable declarada y haz las instrucciones contenidas en el bucle. Después, en cada una de las iteraciones del bucle tendremos en la variable declarada el elemento actual de la expresión. Por tanto, para el caso de los arrays y de las colecciones de objetos, se recorrerá desde el primer elemento que los forma hasta el último.

Observa el contenido del código representado en la siguiente imagen, puedes apreciar cómo se construye un bucle de este tipo y su utilización sobre un array.

Los bucles `for / in` permitirán al programador despreocuparse del número de veces que se ha de iterar, pero no sabremos en qué iteración nos encontramos salvo que se añada artificialmente alguna variable contadora que nos pueda ofrecer esta información.

#### Información

Esta estructura tomará sentido cuando avancemos en el curso y veamos los Arrays y las colecciones de Objetos.

#### 9.4.3. Estructura `while`

El bucle `while` es la primera de las estructuras de repetición controladas por sucesos que vamos a estudiar. La utilización de este bucle responde al planteamiento de la siguiente pregunta: ¿Qué podemos hacer si lo único que sabemos es que se han de repetir un conjunto de instrucciones mientras se cumpla una determinada condición?

La característica fundamental de este tipo de estructura repetitiva estriba en ser útil en aquellos casos en los que las instrucciones que forman el cuerpo del bucle podría ser necesario ejecutarlas o no. Es decir, en el bucle `while` siempre se evaluará la condición que lo controla, y si dicha condición es cierta, el cuerpo del bucle se ejecutará una vez, y se seguirá ejecutando mientras la condición sea cierta. Pero si en la evaluación inicial de la condición ésta no es verdadera, el cuerpo del bucle no se ejecutará.

#### Atención!

Es imprescindible que en el interior del bucle `while` se realice alguna acción que modifique la condición que controla la ejecución del mismo, en caso contrario estaríamos ante un bucle infinito.

#### Sintaxis estructura `while` con una única sentencia:

```
1 while (condición)
2 sentencia;
```

### Sintaxis estructura `while` con un bloque de sentencias:

```

1 while (condición) {
2 sentencia1;
3 ...
4 sentenciaN;
5 }
```

**Funcionamiento:** Mientras la condición sea cierta, el bucle se repetirá, ejecutando la/s instrucción/es de su interior.

En el momento en el que la condición no se cumpla, el control del flujo del programa pasará a la siguiente instrucción que exista justo detrás del bucle `while`.

La condición se evaluará siempre al principio, y podrá darse el caso de que las instrucciones contenidas en él no lleguen a ejecutarse nunca si no se satisface la condición de partida.

#### 9.4.4. Estructura `do while`

La segunda de las estructuras repetitivas controladas por sucesos es `do while`. En este caso, la pregunta que nos planteamos es la siguiente: ¿Qué podemos hacer si lo único que sabemos es que se han de ejecutar, al menos una vez, un conjunto de instrucciones y seguir repitiéndose hasta que se cumpla una determinada condición?.

La característica fundamental de este tipo de estructura repetitiva estriba en ser útil en aquellos casos en los que las instrucciones que forman el cuerpo del bucle necesitan ser ejecutadas, al menos, una vez y repetir su ejecución hasta que la condición sea verdadera. Por tanto, en esta estructura repetitiva siempre se ejecuta el cuerpo del bucle una primera vez.

Es imprescindible que en el interior del bucle se realice alguna acción que modifique la condición que controla la ejecución del mismo, en caso contrario estaríamos ante un bucle infinito.

### Sintaxis estructura `while` con una única sentencia:

```

1 do
2 sentencia;
3 while (condición);
```

### Sintaxis estructura `while` con un bloque de sentencias:

```

1 do {
2 sentencia1;
3 ...
4 sentenciaN;
5 } while (condición);
```

#### Funcionamiento:

El cuerpo del bucle se ejecuta la primera vez, a continuación se evaluará la condición y, si ésta es falsa, el cuerpo el bucle volverá a repetirse. El bucle finalizará cuando la evaluación de la condición sea verdadera.

En ese momento el control del flujo del programa pasará a la siguiente instrucción que exista justo detrás del bucle do-while. La condición se evaluará siempre después de una primera ejecución del cuerpo del bucle, por lo que no se dará el caso de que las instrucciones contenidas en él no lleguen a ejecutarse nunca.



#### 9.4.5. Bucle infinito

Uno de los errores más comunes al implementar cualquier tipo de bucle es que nunca pueda salir, es decir, el bucle se ejecuta durante un número infinito de veces.

Podemos provocarlo intencionadamente como en estos dos ejemplos equivalentes:

##### (NO RECOMENDABLE)

```
1 for(;;){
2 //sentencias
3 }
```

```
1 while(true){
2 //sentencias
3 }
```

O sucede cuando la condición falla por alguna razón, como en el siguiente ejemplo:

## Bucle infinito

```

1 //Programa Java para ilustrar varias trampas de bucles.
2 public class BucleInfinito{
3
4 public static void main(String[] args)
5 {
6 // bucle infinito porque la condición no es apta
7 // la condición; debería haber sido i>0.
8 for (int i = 5; i != 0; i -= 2){
9 System.out.println(i);
10 }
11
12 int x = 5;
13 // bucle infinito porque la actualización
14 // no se proporciona
15 while (x == 5)
16 {
17 System.out.println("En el bucle");
18 }
19 }
20 }
```

Otro inconveniente es que puede estar agregando algo en su objeto de colección a través de un bucle y puede **quedarse sin memoria**. Si intenta ejecutar el siguiente programa, después de un tiempo, se producirá una excepción de falta de memoria. En este ejemplo se hace uso de la colección ArrayList, pero de momento solo necesitamos saber que se comporta como un casillero al que vamos asignando elementos (que evidentemente ocupan memoria)

## Provocar falta de memoria:

```

1 //Programa Java para la excepción de falta de memoria.
2 import java.util.ArrayList;
3 public class HeapSpace
4 {
5 public static void main(String[] args)
6 {
7 ArrayList<Integer> ar = new ArrayList<>();
8 for (int i = 0; i < Integer.MAX_VALUE; i++)
9 {
10 ar.add(i);
11 }
12 }
13 }
```

Salida:

```

1 Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
2 at java.util.Arrays.copyOf(Unknown Source)
3 at java.util.Arrays.copyOf(Unknown Source)
4 at java.util.ArrayList.grow(Unknown Source)
5 at java.util.ArrayList.ensureCapacityInternal(Unknown Source)
6 at java.util.ArrayList.add(Unknown Source)
7 at article.Integer1.main(Integer1.java:9)
```

## 9.5. Estructuras de salto

¿Saltar o no saltar? he ahí la cuestión. En la gran mayoría de libros de programación y publicaciones de Internet, siempre se nos recomienda que prescindamos de sentencias de salto incondicional, es más, se desaconseja su uso por provocar una mala estructuración del código y un incremento en la dificultad para el mantenimiento de los mismos. Pero Java incorpora ciertas sentencias o estructuras de salto que es necesario conocer y que pueden sernos útiles en algunas partes de nuestros programas.

Estas estructuras de salto corresponden a las sentencias `break`, `continue`, las etiquetas de salto y la sentencia `return`. Pasamos ahora a analizar su sintaxis y funcionamiento.

### 9.5.1. Sentencias `break` y `continue`

Se trata de dos instrucciones que permiten modificar el comportamiento de otras estructuras o sentencias de control, simplemente por el hecho de estar incluidas en algún punto de su secuencia de instrucciones.

La sentencia `break` incidirá sobre las estructuras de control `switch`, `while`, `for` y `do while` del siguiente modo:

- Si aparece una sentencia `break` dentro de la secuencia de instrucciones de cualquiera de las estructuras mencionadas anteriormente, dicha estructura terminará inmediatamente.
- Si aparece una sentencia `break` dentro de un bucle anidado sólo finalizará la sentencia de iteración más interna, el resto se ejecuta de forma normal.

Es decir, que `break` sirve para romper el flujo de control de un bucle, aunque no se haya cumplido la condición del bucle. Si colocamos un `break` dentro del código de un bucle, cuando se alcance el `break`, automáticamente se saldrá del bucle pasando a ejecutarse la siguiente instrucción inmediatamente después de él.

La sentencia `continue` incidirá sobre las sentencias o estructuras de control `while`, `for` y `do while` del siguiente modo:

- Si aparece una sentencia `continue` dentro de la secuencia de instrucciones de cualquiera de las sentencias anteriormente indicadas, dicha sentencia dará por terminada la iteración actual y se ejecuta una nueva iteración, evaluando de nuevo la expresión condicional del bucle.
- Si aparece en el interior de un bucle anidado solo afectará a la sentencia de iteración más interna, el resto se ejecutaría de forma normal.

Es decir, la sentencia `continue` forzará a que se ejecute la siguiente iteración del bucle, sin tener en cuenta las instrucciones que pudiera haber después del `continue`, y hasta el final del código del bucle.

### 9.5.2. Etiquetas de salto



Los saltos incondicionales y en especial, saltos a una etiqueta son totalmente **desaconsejables**.

Java permite asociar etiquetas cuando se va a realizar un salto. De este modo puede conseguirse algo más de legibilidad en el código.

Las estructuras de salto `break` y `continue`, pueden tener asociadas etiquetas. Es a lo que se llama un `break` etiquetado o un `continue` etiquetado. Pero sólo se recomienda su uso cuando se hace necesario salir de bucles anidados hacia diferentes niveles. ¿Y cómo se crea un salto a una etiqueta? En primer lugar, crearemos la etiqueta mediante un identificador seguido de dos puntos (`:`). A continuación, se escriben las sentencias Java asociadas a dicha etiqueta encerradas entre llaves. Por así decirlo, la creación de una etiqueta es como fijar un punto de salto en el programa para poder saltar a él desde otro lugar de dicho programa.

¿Cómo se lleva a cabo el salto? Es sencillo, en el lugar donde vayamos a colocar la sentencia `break` o `continue`, añadiremos detrás el identificador de la etiqueta. Con ello, conseguiremos que el salto se realice a un lugar determinado.

La sintaxis será:

```
1 break <etiqueta>;
```



Quizá a aquellos/as que han programado en HTML les suene esta herramienta, ya que tiene cierta similitud con las anclas que pueden crearse en el interior de una página web, a las que nos llevará el hiperenlace o link que hayamos asociado. También para aquellos/as que han creado alguna vez archivos por lotes o archivos batch bajo MSDOS es probable que también les resulte familiar el uso de etiquetas, pues la sentencia GOTO que se utilizaba en este tipo de archivos, hacía saltar el flujo del programa al lugar donde se ubicaba la etiqueta que se indicara en dicha sentencia.

### 9.5.3. return

Ya sabemos cómo modificar la ejecución de bucles y estructuras condicionales múltiples, pero ¿Podríamos modificar la ejecución de un método? ¿Es posible hacer que éstos detengan su ejecución antes de que finalice el código asociado a ellos?. Sí es posible, a través de la sentencia `return` podremos conseguirlo. La sentencia `return` puede utilizarse de dos formas:

- Para terminar la ejecución del método donde esté escrita, con lo que transferirá el control al punto desde el que se hizo la llamada al método, continuando el programa por la sentencia inmediatamente posterior.
- Para devolver o retornar un valor, siempre que junto a `return` se incluya una expresión de un tipo determinado. Por tanto, en el lugar donde se invocó al método se obtendrá el valor resultante de la evaluación de la expresión que acompañaba al método.

### Importante

En general, una sentencia `return` suele aparecer al final de un método, de este modo el método tendrá una entrada y una salida. También es posible utilizar una sentencia `return` en cualquier punto de un método, con lo que éste finalizará en el lugar donde se encuentre dicho `return`. No será recomendable incluir más de un `return` en un método y por regla general, deberá ir al final del método como hemos comentado.

El valor de retorno es opcional, si lo hubiera debería de ser del mismo tipo o de un tipo compatible al tipo del valor de retorno definido en la cabecera del método, pudiendo ser desde un entero a un objeto creado por nosotros. Si no lo tuviera, el tipo de retorno sería `void`, y `return` serviría para salir del método sin necesidad de llegar a ejecutar todas las instrucciones que se encuentran después del `return`.

## 9.6. Excepciones

A lo largo de nuestro aprendizaje de Java nos hemos topado en alguna ocasión con errores, pero éstos suelen ser los que nos ha indicado el compilador. Un punto y coma por aquí, un nombre de variable incorrecto por allá, pueden hacer que nuestro compilador nos avise de estos descuidos.

Cuando los vemos, se corrigen y obtenemos nuestra clase compilada correctamente.

Pero, ¿Sólo existen este tipo de errores? ¿Podrían existir errores no sintácticos en nuestros programas?. Está claro que sí, un programa perfectamente compilado en el que no existen errores de sintaxis, puede generar otros tipos de errores que quizás aparezcan en tiempo de ejecución. A estos errores se les conoce como **excepciones**.

Aprenderemos a gestionar de manera adecuada estas excepciones y tendremos la oportunidad de utilizar el potente sistema de manejo de errores que Java incorpora. La potencia de este sistema de manejo de errores radica en:

1. Que el código que se encarga de manejar los errores, es perfectamente identificable en los programas. Este código puede estar separado del código que maneja la aplicación.
2. Que Java tiene una gran cantidad de errores estándar asociados a multitud de fallos comunes, como por ejemplo divisiones por cero, fallos de entrada de datos, etc. Al tener tantas excepciones localizadas, podemos gestionar de manera específica cada uno de los errores que se produzcan.

En Java se pueden preparar los fragmentos de código que pueden provocar errores de ejecución para que si se produce una excepción, el flujo del programa es lanzado (`throw`) hacia ciertas zonas o rutinas que han sido creadas previamente por el programador y cuya finalidad será el tratamiento efectivo de dichas excepciones. Si no se captura la excepción, el programa se detendrá con toda probabilidad.

En Java, las excepciones están representadas por clases. El paquete `java.lang.Exception` y sus subpaquetes contienen todos los tipos de excepciones. Todas las excepciones derivarán de la clase `Throwable`, existiendo clases más específicas. Por debajo de la clase `Throwable` existen las clases `Error` y `Exception`. `Error` es una clase que se encargará de los errores que se produzcan en la máquina virtual, no en nuestros programas. Y la clase `Exception` será la que a nosotros nos interese conocer, pues gestiona los errores provocados en los programas.

Java lanzará una excepción en respuesta a una situación poco usual. Cuando se produce un error se genera un objeto asociado a esa excepción. Este objeto es de la clase `Exception` o de alguna de sus herederas. Este objeto se pasa al código que se ha definido para manejar la excepción. Dicho código puede manipular las propiedades del objeto `Exception`.

El programador también puede lanzar sus propias excepciones. Las excepciones en Java serán objetos de clases derivadas de la clase base `Exception`. Existe toda una jerarquía de clases derivada de la clase base `Exception`. Estas clases derivadas se ubican en dos grupos principales:

- Las excepciones en tiempo de ejecución, que ocurren cuando el programador no ha tenido cuidado al escribir su código.
- Las excepciones que indican que ha sucedido algo inesperado o fuera de control.

En la siguiente imagen te ofrecemos una aproximación a la jerarquía de las excepciones en Java.

```
classDiagram
 class Object
 class Throwable
 Object <|-- Throwable
 namespace Comprobadas_Checked {
 class Exception
 class IOException
 class UsersExceptions
 class Other1[...]
 class Other2[...]
 class Other3[...]
```

```

}
namespace No_Comprobadas_Unchecked {
 class RuntimeException
 class Error
 class ArithmeticException
 class ArrayIndexOutOfBoundsException
 class Other4[...]
 }
 Exception <|-- Other1
 Exception <|-- IOException
 IOException <|-- Other2
 IOException <|-- Other3
 Exception <|-- UsersExceptions
 Exception <|-- RuntimeException
 RuntimeException <|-- ArithmeticException
 RuntimeException <|-- ArrayIndexOutOfBoundsException
 RuntimeException <|-- Other4
 Throwable <|-- Exception
 Throwable <|-- Error
}

```

Y aquí tenemos una lista de las más habituales con su explicación:

| NOMBRE                                | DESCRIPCIÓN                                                                                                             |
|---------------------------------------|-------------------------------------------------------------------------------------------------------------------------|
| <b>FileNotFoundException</b>          | Lanza una excepción cuando el fichero no se encuentra.                                                                  |
| <b>ClassNotFoundException</b>         | Lanza una excepción cuando no existe la clase.                                                                          |
| <b>EOFException</b>                   | Lanza una excepción cuando llega al final del fichero.                                                                  |
| <b>ArrayIndexOutOfBoundsException</b> | Lanza una excepción cuando se accede a una posición de un array que no exista.                                          |
| <b>NumberFormatException</b>          | Lanza una excepción cuando se procesa un numero pero este es un dato alfanumérico.                                      |
| <b>NullPointerException</b>           | Lanza una excepción cuando intentando acceder a un miembro de un objeto para el que todavía no hemos reservado memoria. |
| <b>IOException</b>                    | Generaliza muchas excepciones anteriores. La ventaja es que no necesitamos controlar cada una de las excepciones.       |
| <b>Exception</b>                      | Es la clase padre de IOException y de otras clases. Tiene la misma ventaja que IOException.                             |
| <b>ArithmaticException</b>            | Se lanza por ejemplo, cuando intentamos dividir un número entre cero.                                                   |

#### 9.6.1. El manejo de excepciones

Como hemos comentado, siempre debemos controlar las excepciones que se puedan producir o de lo contrario nuestro software quedará expuesto a fallos. Las excepciones pueden tratarse de dos formas:

- **Interrupción.** En este caso se asume que el programa ha encontrado un error irrecuperable. La operación que dio lugar a la excepción se anula y se entiende que no hay manera de regresar al código que provocó la excepción. Es decir, la operación que dio originó el error, se anula.
- **Reanudación.** Se puede manejar el error y regresar de nuevo al código que provocó el error.

Java emplea la primera forma, pero puede simularse la segunda mediante la utilización de un bloque `try` en el interior de un `while`, que se repetirá hasta que el error deje de existir. En la sección de ejemplos de puedes ver como poner el `try-catch` dentro de un `do while`.

#### 9.6.2. Capturar una excepción

Para poder capturar excepciones, emplearemos la estructura de captura de excepciones `try-catch-finally`.

Básicamente, para capturar una excepción lo que haremos será declarar bloques de código donde es posible que ocurra una excepción. Esto lo haremos mediante un bloque `try` (intentar). Si ocurre una excepción dentro de estos bloques, se lanza una excepción. Estas excepciones lanzadas se pueden capturar por medio de bloques `catch`. Será dentro de este tipo de bloques donde se hará el manejo de las excepciones.

### Sintaxis try-catch :

```

1 try {
2 //código que puede generar excepciones;
3 } catch (Tipo_excepcion_1 objeto_excepcion) {
4 //Manejo de excepción de Tipo_excepcion_1;
5 } catch (Tipo_excepcion_2 objeto_excepcion) {
6 //Manejo de excepción de Tipo_excepcion_2;
7 }
8 ...
9 finally {
10 //instrucciones que se ejecutan siempre
11 }
```

En esta estructura, la parte `catch` puede repetirse tantas veces como excepciones diferentes se deseen capturar. La parte `finally` es opcional y, si aparece, solo podrá hacerlo una vez.

Cada `catch` maneja un tipo de excepción. Cuando se produce una excepción, se busca el `catch` que posea el manejador de excepción adecuado, será el que utilice el mismo tipo de excepción que se ha producido. Esto puede causar problemas si no se tiene cuidado, ya que la clase `Exception` es la superclase de todas las demás. Por lo que si se produjo, por ejemplo, una excepción de tipo `Aritmetic Exception` y el primer `catch` captura el tipo genérico `Exception`, será ese `catch` el que se ejecute y no los demás.

Por eso el último `catch` debe ser el que capture excepciones genéricas y los primeros deben ser los más específicos. Lógicamente si vamos a tratar a todas las excepciones (sean del tipo que sean) igual, entonces basta con un solo `catch` que capture objetos `Exception`.

### Recuerda

En Java, cuando un bloque de código puede provocar una excepción pero no se maneja adecuadamente, se produce lo que se conoce como una "excepción no controlada" o "excepción no capturada". Cuando ocurre una excepción no controlada, Java sigue un conjunto de reglas específicas para manejarla:

- Propagación de excepciones:** Java busca en la pila de llamadas (el seguimiento de la ejecución del programa) para ver si el método actual maneja la excepción. Si el método actual no maneja la excepción, la excepción se "propaga" hacia arriba en la pila de llamadas. (Piensa en una burbuja de aire en el fondo del mar intentando buscar una salida)
- Búsqueda de un manejador de excepciones:** La excepción propagada continúa buscando un manejador de excepciones adecuado a medida que se retrocede a través de los métodos que llamaron al método actual. Si se encuentra un bloque `try-catch` que puede manejar la excepción, se ejecutará el código del bloque `catch` correspondiente.
- Si no se encuentra un manejador adecuado:** Si la excepción llega a la parte superior de la pila de llamadas y no se encuentra un manejador de excepciones adecuado, el programa se detendrá y se imprimirá un mensaje de error en la consola, que contiene información sobre la excepción, como su tipo, mensaje y seguimiento de pila (`stack trace`).

#### 9.6.3. Delegación de excepciones con `throws`

¿Puede haber problemas con las excepciones al usar llamadas a métodos en nuestros programas? Efectivamente, si se produjese una excepción es necesario saber quién será el encargado de solucionarla. Puede ser que sea el propio método llamado o el código que hizo la llamada a dicho método.

Quizá pudieramos pensar que debería ser el propio método el que se encargue de sus excepciones, aunque es posible hacer que la excepción sea resuelta por el código que hizo la llamada. Cuando un método utiliza una sentencia que puede generar una excepción, pero dicha excepción no es capturada y tratada por él, sino que se encarga su gestión a quién llamó al método, decimos que se ha producido delegación de excepciones.

Para establecer esta delegación, en la cabecera del método se declara el tipo de excepciones que puede generar y que deberán ser gestionadas por quien invoque a dicho método. Utilizaremos para ello la sentencia `throws` y tras esa palabra se indica qué excepciones puede provocar el código del método. Si ocurre una excepción en el método, el código abandona ese método y regresa al código desde el que se llamó al método. Allí se buscará el `catch` apropiado para esa excepción. Su sintaxis es la siguiente:

### Sintaxis throws :

```

1 public class Delegacion_Excepciones {
2 ...
3 public int leeAnio(BufferedReader lector) throws IOException, NumberFormatException{
4 String linea = teclado.readLine();
5 return Integer.parseInt(linea);
6 }
7 ...
8 }
```

Donde `IOException` y `NumberFormatException`, serían dos posibles excepciones que el método `leeAnio` podría generar, pero que no gestiona. Por tanto, un método puede incluir en su cabecera un listado de excepciones que puede lanzar, separadas por comas

#### 9.6.4. Crear y lanzar excepciones de usuario

Las excepciones de usuario son subclases de la clase `Exception` que podemos crear y lanzar en nuestros programas para avisar sobre determinadas situaciones.

##### 9.6.4.1. CREAR UNA NUEVA EXCEPCIÓN

Para crear una nueva excepción tenemos que crear una clase derivada (subclase) de la clase `Exception`.

La clase `Exception` tiene dos constructores, uno sin parámetros y otro que acepta un `String` con un texto descriptivo de la excepción. Todas las excepciones de usuario las crearemos de la siguiente forma:

### Constructores de Exception:

```

1 class NombreExcepcion extends Exception {
2 public NombreExcepcion(){
3 super();
4 }
5 public NombreExcepcion(String msg){
6 super(msg);
7 }
8 }
```

##### 9.6.4.2. LANZAR UNA EXCEPCIÓN

Las excepciones se lanzan mediante la instrucción `throw`. La sintaxis es:

### Instrucción throw :

```
1 throw new NombreExcepcion("Mensaje descriptivo de la situación inesperada");
```

Ya que se tratará de una excepción comprobada, en la cabecera del método que lanza la excepción habrá que propagarla.

#### 9.6.5. Excepciones Checked y unChecked

En Java, las excepciones se dividen en dos categorías principales: excepciones "checked" (comprobadas) y excepciones "unchecked" (no comprobadas).

- 1. Excepciones Comprobadas (Checked Exceptions):** - Las excepciones comprobadas son aquellas que el compilador obliga a manejar. Esto significa que, si un método puede lanzar una excepción comprobada, el programador está obligado a manejarla de alguna manera, ya sea mediante la declaración del método con `throws` o mediante el manejo directo con un bloque `try-catch`. - Ejemplos de excepciones comprobadas incluyen `IOException` y `SQLException`. - Estas excepciones suelen representar situaciones en las que un programa no puede continuar normalmente y se espera que el código las maneje de manera adecuada.

### Excepción comprobada:

```

1 import java.io.FileReader;
2 import java.io.FileNotFoundException;
3
4 public class EjemploCheckedException {
5 public static void main(String[] args) {
6 try {
7 FileReader file = new FileReader("archivo.txt");
8 } catch (FileNotFoundException e) {
9 System.out.println("Archivo no encontrado: " + e.getMessage());
10 }
11 }
12 }
```

- 1. Excepciones No Comprobadas (Unchecked Exceptions):** - Las excepciones no comprobadas son aquellas que el compilador no requiere que se manejen explícitamente. Normalmente, son subclases de `RuntimeException`. - Estas excepciones suelen deberse a errores de programación, como acceder a un índice fuera de los límites de un array (`ArrayIndexOutOfBoundsException`) o intentar convertir un objeto a un tipo incompatible (`ClassCastException`). - Aunque no se requiere que se manejen explícitamente, es buena práctica manejarlas para evitar que el programa termine abruptamente.

### Excepción no comprobada:

```

1 public class EjemploUncheckedException {
2 public static void main(String[] args) {
3 int[] numeros = {1, 2, 3};
4 System.out.println(numeros[4]); // Esto lanzará ArrayIndexOutOfBoundsException
5 }
6 }
```

#### 9.6.5.1. ¿COMO SÉ SI UNA EXCEPCIÓN ES DE UN TIPO O DE OTRO?

La principal diferencia radica en la obligación del compilador de manejar o declarar excepciones. Las excepciones comprobadas deben ser manejadas o declaradas en el código, mientras que las excepciones no comprobadas no tienen esta obligación y generalmente se deben a errores de programación.

En Java, puedes distinguir entre excepciones comprobadas y no comprobadas principalmente por el tipo de clase que heredan. Aquí hay algunas pautas generales:

- 1. Excepciones Comprobadas (Checked Exceptions):** - Las excepciones comprobadas suelen ser subclases directas de la clase `Exception` (o alguna de sus subclases), pero no heredan de `RuntimeException` ni de sus subclases. - Ejemplos comunes incluyen `IOException`, `SQLException`, y cualquier excepción que herede directamente de `Exception` (pero no de `RuntimeException`).
- 2. Excepciones No Comprobadas (Unchecked Exceptions):** - Las excepciones no comprobadas suelen ser subclases directas de la clase `RuntimeException`. - Ejemplos comunes incluyen `NullPointerException`, `ArrayIndexOutOfBoundsException`, y cualquier excepción que herede directamente de `RuntimeException`.

### Importante

Ten en cuenta que estas son pautas generales y puede haber excepciones personalizadas o situaciones específicas en las que estas reglas no se apliquen estrictamente. Para obtener información precisa sobre un tipo de excepción específico, puedes consultar la documentación de Java o examinar la jerarquía de clases y herencia de la excepción en cuestión.

## 9.7. Aserciones ( Assertions )

Una aserción (afirmación) permite probar la exactitud de cualquier suposición que se haya hecho en el programa. Una afirmación se logra utilizando la declaración de `assertion` en Java. Al ejecutar una aserción, se cree que es cierta. Si falla, JVM genera un error denominado `AssertionError`. Se utiliza principalmente con fines de prueba durante el desarrollo.

La declaración de afirmación se usa con una expresión booleana y se puede escribir de dos maneras diferentes.

**Primera forma:**

```
1 assert expression;
```

**Segunda forma:**

```
1 assert expression1 : expression2;
```

**Ejemplo:**

```
1 import java.util.Scanner;
2
3 public class P7_Assertions {
4 // Programa Java para demostrar el uso de las assertions
5 public static void main(String[] args) {
6 Scanner entrada = new Scanner(System.in);
7 System.out.print("Introduce tu edad: ");
8 int age = entrada.nextInt();
9 assert (age >= 18) : "No puede votar";
10 System.out.println("La edad del votante es de " + age);
11 }
12 }
```

Salida sin assertions:

```
1 Introduce tu edad: 14
2 La edad del votante es de 14
```

Después de habilitar las assertions:

**Saber más...**

Puedes habilitar las assertions añadiendo los parámetros de la JVM en IntelliJ:

- `-ea`: Enable Assertions (habilitar aserciones)
- `-da`: Disable Assertions (deshabilitar aserciones, que es la opción por defecto)

Puedes consultar este enlace para saber donde agregar estas opciones: <https://stackoverflow.com/questions/68848158/java-assertions-in-intellij-idea-community>

Salida:

```
1 Introduce tu edad: 14
2 Exception in thread "main" java.lang.AssertionError: No puede votar
3 at UD03.P7_Assertions.main(P7_Assertions.java:11)
```

**Otro ejemplo:**

```
1 package UD03;
2
3 public class P7_Assertions2 {
4 public static void main(String[] args) {
5 System.out.println("Probando Aserciones...");
6 assert true : "Neveremos esto.";
7 assert false : "Esto solo lo veremos si activamos las aserciones.";
8 }
9 }
```

Ejecución sin aserciones:

```
1 Probando Aserciones...
```

Y con aserciones:

```
1 Probando Aserciones...
2 Exception in thread "main" java.lang.AssertionError: Esto solo lo veremos si activamos las aserciones.
3 at UD03.P7_Assertions2.main(P7_Assertions2.java:7)
```

### 9.7.1. ¿Por qué utilizar aserciones?

Dondequiero que un programador quiera ver si sus suposiciones son erróneas o no.

- Para asegurarse de que un código que parece inalcanzable sea realmente inalcanzable.
- Para asegurarse de que las suposiciones escritas en los comentarios sean correctas.
- Para asegurarse de que no se alcance el caso default del switch.
- Para comprobar el estado del objeto.
- Al comienzo del método.
- Despues de la invocación del método.

### 9.7.2. Aserción o Excepciones

Las aserciones se utilizan principalmente para comprobar situaciones lógicamente imposibles. Por ejemplo, se pueden utilizar para comprobar el estado que espera un código antes de empezar a ejecutarse o el estado después de que termine de ejecutarse. A diferencia del manejo normal de excepciones/errores, las aserciones generalmente están deshabilitadas en tiempo de ejecución.

¿Dónde utilizarlas?:

- Argumentos para los métodos privados. Los argumentos para los métodos privados los proporciona únicamente el código del desarrollador y es posible que este desee comprobar sus suposiciones sobre los argumentos.
- Casos condicionales.
- Condicones al inicio de cualquier método.

¿Dónde **NO** utilizar aserciones?:

- Las aserciones no deben usarse para reemplazar mensajes de error
- Las aserciones no deben usarse para verificar argumentos en los métodos públicos, ya que pueden ser proporcionados por el usuario.
- Para manejar los errores proporcionados por los usuarios usaremos las excepciones.
- Las aserciones no deben usarse en argumentos de línea de comando.

## 9.8. Ejemplos UD03

### 9.8.1. if e if-else

Para completar la información que debes saber sobre las estructuras `if` e `if-else`, observa el siguiente código. En él podrás analizar el programa que realiza el cálculo de la nota de un examen de tipo test. Además de calcular el valor de la nota, se ofrece como salida la calificación no numérica de dicho examen. Para obtenerla, se combinarán las diferentes estructuras condicionales aprendidas hasta ahora.

Presta especial atención a los comentarios incorporados en el código fuente, así como a la forma de combinar las estructuras condicionales y a las expresiones lógicas utilizadas en ellas.

```

1 package UD03;
2 public class Sentencias_Condicionales {
3 /*Vamos a realizar el cálculo de la nota de un examen
4 * de tipo test. Para ello, tendremos en cuenta el número
5 * total de pregunta, los aciertos y los errores. Dos errores
6 * anulan una respuesta correcta.
7 *
8 * Finalmente, se muestra por pantalla la nota obtenida, así
9 * como su calificación no numérica.
10 *
11 * La obtención de la calificación no numérica se ha realizado
12 * combinando varias estructuras condicionales, mostrando expresiones
13 * lógicas compuestas, así como anidamiento.
14 */
15 public static void main(String[] args) {
16 // Declaración e inicialización de variables
17 int num_aciertos = 12;
18 int num_errores = 3;
19 int num_preguntas = 20;
20 float nota = 0;
21 String calificacion = "";
22 //Procesamiento de datos
23 nota = ((num_aciertos - (num_errores / 2)) * 10) / num_preguntas;
24
25 if (nota < 5) {
26 calificacion = "INSUFICIENTE";
27 } else {
28 /* Cada expresión lógica de estos if está compuesta por dos
29 * expresiones lógicas combinadas a través del operador Y o AND
30 * que se representa con el símbolo &&. De tal manera, que para
31 * que la expresión lógica se cumpla (sea verdadera) la variable
32 * nota debe satisfacer ambas condiciones simultáneamente
33 */
34 if (nota >= 5 && nota < 6) {
35 calificacion = "SUFICIENTE";
36 } else if (nota >= 6 && nota < 7) {
37 calificacion = "BIEN";
38 } else if (nota >= 7 && nota < 9) {
39 calificacion = "NOTABLE";
40 } else if (nota >= 9 && nota <= 10) {
41 calificacion = "SOBRESALIENTE";
42 }
43 }
44 //Salida de información
45 System.out.println("La nota obtenida es: " + nota);
46 System.out.println("y la calificación obtenida es: " + calificacion);
47 }
48 }
```

### 9.8.2. switch

Comprueba el siguiente fragmento de código en el que se resuelve el cálculo de un examen de tipo test, utilizando la estructura `switch`.

```
1 package UD03;
2
3 public class P3_2_condicional_switch {
4
5 /*
6 * Vamos a realizar el cálculo de la nota de un examen de tipo test. Para
7 * ello, tendremos en cuenta el número total de preguntas, los aciertos y
8 * los errores. Dos errores anulan una respuesta correcta.
9 *
10 * La nota que vamos a obtener será un número entero.
11 *
12 * Finalmente, se muestra por pantalla la nota obtenida, así como su
13 * calificación no numérica.
14 *
15 * La obtención de la calificación no numérica se ha realizado utilizando la
16 * estructura condicional múltiple o switch.
17 *
18 */
19 public static void main(String[] args) {
20 // Declaración e inicialización de variables
21 int num_aciertos = 17;
22 int num_errores = 3;
23 int num_preguntas = 20;
24 int nota = 0;
25 String calificacion = "";
26 //Procesamiento de datos
27 nota = ((num_aciertos - (num_errores / 2)) * 10) / num_preguntas;
28 switch (nota) {
29 case 5:
30 calificacion = "SUFICIENTE";
31 break;
32 case 6:
33 calificacion = "BIEN";
34 break;
35 case 7:
36 calificacion = "NOTABLE";
37 break;
38 case 8:
39 calificacion = "NOTABLE";
39 break;
40 case 9:
41 calificacion = "SOBRESALIENTE";
42 break;
43 case 10:
44 calificacion = "SOBRESALIENTE";
45 break;
46 default:
47 calificacion = "INSUFICIENTE";
48 }
49 //Salida de información
50 System.out.println("La nota obtenida es: " + nota);
51 System.out.println("y la calificación obtenida es: " + calificacion);
52 }
```

```

1 //Expresiones switch mejoradas JAVA 12
2 int entero = 5;
3
4 String numericString = switch (entero) {
5 case 0 -> "cero";
6 case 1, 3, 5, 7, 9 -> "impar";
7 case 2, 4, 6, 8, 10 -> "par";
8 default -> "error";
9 };
10 System.out.println(numericString); //impar
11
12 //Expresiones switch mejoradas JAVA 13
13
14 int entero2 = 4;
15
16 String numericString2 = switch (entero2) {
17 case 0 -> {
18 String value = calculaCero();
19 yield value;
20 }
21 case 1, 3, 5, 7, 9 -> {
22 String value = calculaImpar();
23 yield value;
24 }
25
26 case 2, 4, 6, 8, 10 -> {
27 String value = calculaPar();
28 yield value;
29 }
30
31 default -> {
32 String value = calculaDefecto();
33 yield value;
34 }
35 };
36 System.out.println(numericString); //calculaPar()
37 }
38 static String calculaCero() {return "";}
39 static String calculaImpar() {return "";}
40 static String calculaPar() {return "";}
41 static String calculaDefecto() {return "";}
42 }
```

### 9.8.3. for

Observa el siguiente archivo Java y podrás analizar un ejemplo de utilización del bucle for para la impresión por pantalla de la tabla de multiplicar del siete. Lee atentamente los comentarios incluidos en el código, pues aclaran algunas cuestiones interesantes sobre este bucle.

```

1 package UD03;
2
3 public class Repetitiva_For {
4 /* En este ejemplo se utiliza la estructura repetitiva for
5 * para representar en pantalla la tabla de multiplicar del siete
6 */
7 public static void main(String[] args) {
8 // Declaración e inicialización de variables
9 int numero = 7;
10 int contador;
11 int resultado = 0;
12 //Salida de información
13 System.out.println("Tabla de multiplicar del " + numero);
14 System.out.println(".....");
15 //Utilizamos ahora el bucle for
16 for (contador = 1; contador <= 10; contador++) {
17 /* La cabecera del bucle incorpora la inicialización de la variable
18 * de control, la condición de multiplicación hasta el 10 y el
19 * incremento de dicha variable de uno en uno en cada iteración del
20 * bucle.
21 * En este caso contador++ incrementará en una unidad el valor de
22 * dicha variable.
23 */
24 resultado = contador * numero;
25 System.out.println(numero + " x " + contador + " = " + resultado);
26 /* A través del operador + aplicado a cadenas de caracteres,
27 * concatenamos los valores de las variables con las cadenas de
28 * caracteres que necesitamos para representar correctamente la
29 * salida de cada multiplicación.
30 */
31 }
32 }
33 }
```

#### 9.8.4. while

Observa el siguiente código java y podrás analizar un ejemplo de utilización del bucle `while` para la impresión por pantalla de la tabla de multiplicar del siete. Lee atentamente los comentarios incluidos en el código, pues aclaran algunas cuestiones interesantes sobre este bucle. Como podrás comprobar, el resultado de este bucle es totalmente equivalente al obtenido utilizando el bucle `for`.

```

1 package UD03;
2
3 public class Repetitiva_While {
4
5 public static void main(String[] args) {
6 // Declaración e inicialización de variables
7 int numero = 7;
8 int contador;
9 int resultado = 0;
10 //Salida de información
11 System.out.println("Tabla de multiplicar del " + numero);
12 System.out.println(".....");
13 //Utilizamos ahora el bucle while
14 contador = 1; //inicializamos la variable contadora
15 while (contador <= 10){ //Establecemos la condición del bucle
16 resultado = contador * numero;
17 System.out.println(numero + " x " + contador + " = " + resultado);
18 //Modificamos el valor de la variable contadora, para hacer que el
19 //bucle pueda seguir iterando hasta llegar a finalizar
20 contador++;
21 }
22 }
23 }
```

#### 9.8.5. do while

Ahora podrás analizar un ejemplo de utilización del bucle `do while` para la impresión por pantalla de la tabla de multiplicar del siete. Lee atentamente los comentarios incluidos en el código, pues aclaran algunas cuestiones interesantes sobre este bucle. Como podrás comprobar, el resultado de este bucle es totalmente equivalente al obtenido utilizando el bucle `for` y el bucle `while`.

```

1 package UD03;
2
3 public class Repetitiva_DoWhile {
4
5 public static void main(String[] args) {
6 // Declaración e inicialización de variables
7 int numero = 7;
8 int contador;
9 int resultado = 0;
10 //Salida de información
11 System.out.println("Tabla de multiplicar del " + numero);
12 System.out.println(".....");
13 //Utilizamos ahora el bucle do-while
14 contador = 1; //inicializamos la variable contadora
15 do {
16 resultado = contador * numero;
17 System.out.println(numero + " x " + contador + " = " + resultado);
18 //Modificamos el valor de la variable contadora, para hacer que el
19 //bucle pueda seguir iterando hasta llegar a finalizar
20 contador++;
21 } while (contador <= 10); //Establecemos la condición del bucle
22 }
23 }
```

#### 9.8.6. break

Aunque no es recomendable su uso aquí tienes un ejemplo de la estructura `break`

```

1 package UD03;
2
3 public class Sentencia_Break {
4
5 public static void main(String[] args) {
6 int contador;
7 for (contador=1;contador<=10;contador++){
8 if (contador==7){
9 break;
10 System.out.println("Valor: " + contador);
11 }
12 System.out.println("Fin del programa");
13 /*
14 * El bucle solo se ejecutará en 6 ocasiones, ya que cuando
15 * la variable contador sea igual a 7 encontraremos un break que
16 * romperá el flujo del bucle, transfiriéndonos a la sentencia que
17 * imprime el mensaje de Fin del programa.
18 */
19 }
20 }

```

### 9.8.7. continue

Aunque no es recomendable su uso aquí tienes un ejemplo de la estructura `continue`

```

1 package UD03;
2
3 public class Sentencia_Continue {
4
5 public static void main(String[] args) {
6 int contador;
7 System.out.println("Imprimiendo los números pares que hay del 1 al 10...");
8 for (contador = 1; contador <= 10; contador++) {
9 if (contador % 2 != 0) {
10 continue;
11 }
12 System.out.println(contador + " ");
13 }
14 System.out.println("\nFin del programa");
15 /*
16 * Las iteraciones del bucle que generarán la impresión de cada uno de
17 * los números pares, serán aquellas en las que el resultado de calcular
18 * el resto de la división entre 2 de cada valor de la variable
19 * contador, sea igual a 0.
20 */
21 }
22 }

```

### 9.8.8. Etiquetas de salto

A continuación, te ofrecemos un ejemplo de declaración y uso de etiquetas en un bucle. Como podrás apreciar, las sentencias asociadas a cada etiqueta están encerradas entre llaves para delimitar así su ámbito de acción.

```

1 package UD03;
2
3 public class EtiquetasSalto {
4
5 public static void main(String[] args) {
6 for (int i = 1; i < 3; i++) {
7 bloque_uno:
8 {
9 bloque_dos:
10 {
11 System.out.println("Iteración: " + i);
12 if (i == 1) {
13 break bloque_uno;
14 }
15 if (i == 2) {
16 break bloque_dos;
17 }
18 }
19 System.out.println("después del bloque dos");
20 }
21 System.out.println("después del bloque uno");
22 }
23 System.out.println("Fin del bucle");
24 }
25 }

```

### 9.8.9. Sentencia return

En el siguiente archivo java encontrarás el código de un programa que obtiene la suma de dos números, empleando para ello un método sencillo que retorna el valor de la suma de los números que se le han pasado como parámetros. Presta atención a los comentarios y fíjate en las conversiones a entero de la entrada de los operandos por consola.

```

1 package UD03;
2
3 import java.io.*;
4
5 public class Sentencia_Return {
6
7 private static BufferedReader stdin = new BufferedReader(
8 new InputStreamReader(System.in));
9
10 public static int suma(int numero1, int numero2) {
11 int resultado;
12 resultado = numero1 + numero2;
13 return resultado; //Mediante return devolvemos el resultado de la suma
14 }
15
16 public static void main(String[] args) throws IOException {
17 //Declaración de variables
18 String input; //Esta variable recibirá la entrada de teclado
19 int primer_numero, segundo_numero; //Estas variables almacenarán los operandos
20 // Solicitamos que el usuario introduzca dos números por consola
21 System.out.print("Introduce el primer operando:");
22 input = stdin.readLine(); //Leemos la entrada como cadena de caracteres
23 primer_numero = Integer.parseInt(input); //Transformamos a entero lo introducido
24 System.out.print("Introduce el segundo operando: ");
25 input = stdin.readLine(); //Leemos la entrada como cadena de caracteres
26 segundo_numero = Integer.parseInt(input); //Transformamos a entero lo introducido
27 //Imprimimos los números introducidos
28 System.out.println("Los operandos son: " + primer_numero + " y " + segundo_numero);
29 System.out.println("obteniendo su suma... ");
30 //Invocamos al método que realiza la suma, pasándole los parámetros adecuados
31 System.out.println("La suma de ambos operandos es: " +
32 suma(primer_numero, segundo_numero));
33 }
34 }
```

### 9.8.10. Excepciones

Vamos a realizar un programa en Java en el que se solicite al usuario la introducción de un número por teclado comprendido entre el 0 y el 100. Utilizando manejo de excepciones, controlaremos la entrada de dicho número y volver a solicitarlo en caso de que ésta sea incorrecta.

```

1 package UD03;
2
3 import java.io.*;
4 import java.util.Scanner;
5
6 public class P6_1_Excepciones {
7
8 public static void main(String[] args) {
9 int numero = -1;
10 int intentos = 0;
11 String linea;
12 Scanner teclado = new Scanner(System.in);
13 do {
14 try {
15 System.out.print("Introduzca un número entre 0 y 100: ");
16 linea = teclado.nextLine();
17 numero = Integer.parseInt(linea);
18 } catch (NumberFormatException e) {
19 System.out.println("Debe introducir un número entre 0 y 100.");
20 } catch (Exception e) {
21 System.out.println("Error al leer del teclado.");
22 } finally {
23 intentos++;
24 }
25 } while (numero < 0 || numero > 100);
26 System.out.println("El número introducido es: " + numero);
27 System.out.println("Número de intentos: " + intentos);
28 }
29 }
```

En este programa se solicita repetidamente un número utilizando una estructura `do while`, mientras el número introducido sea menor que 0 y mayor que 100. Como al solicitar el número pueden producirse los errores siguientes:

- De entrada de información a través de la excepción `Exception` generada por el método `nextLine()` de la clase `Scanner`.
- De conversión de tipos a través de la excepción `NumberFormatException` generada por el método `parseInt()`.

Entonces se hace necesaria la utilización de bloques `catch` que gestionen cada una de las excepciones que puedan producirse. Cuando se produce una excepción, se compara si coincide con la excepción del primer `catch`. Si no coincide, se compara con la del segundo `catch` y así sucesivamente. Si se encuentra un `catch` que coincide con la excepción a gestionar, se ejecutará el bloque de sentencias asociado a éste.

Si ningún bloque `catch` coincide con la excepción lanzada, dicha excepción se lanzará fuera de la estructura `try-catch-finally`.

El bloque `finally`, se ejecutará tanto si `try` terminó correctamente, como si se capturó una excepción en algún bloque `catch`. Por tanto, si existe bloque `finally` éste se ejecutará siempre.

#### 9.8.10.1. EJEMPLO DE LA PROPAGACIÓN DE EXCEPCIONES

Aquí tienes este otro ejemplo para comprender cómo se propaga una excepción hacia arriba en la pila de ejecución en Java

```

1 package UD03;
2
3 import java.util.Scanner;
4
5 public class P6_2_PropagacionExcepciones {
6 public static void main(String[] args) {
7 Scanner teclado = new Scanner(System.in);
8 try {
9 System.out.print("Introduzca un número entre 0 y 100: ");
10 String linea = teclado.nextLine();
11 int numero = Integer.parseInt(linea);
12 metodoA(numero);
13 } catch (Exception e) {
14 System.out.println("Excepción atrapada en el método main: " + e.getMessage());
15 }
16 }
17
18 public static void metodoA(int numero) {
19 try {
20 metodoB(numero);
21 } catch (ArithmeticException e) {
22 System.out.println("Excepción atrapada en el método A: " + e.getMessage());
23 }
24 }
25
26 public static void metodoB(int divisor) {
27 int resultado = 10 / divisor;
28 }
29}
30

```

#### 9.8.10.2. EJEMPLO DE EXCEPCIÓN PERSONALIZADA

En este ejemplo definimos nuestra propia clase de Excepciones (en una clase interna), la lanzamos en un método de manera personalizada, y posteriormente capturamos la excepción en el método `main`.

```

1 import java.util.Scanner;
2
3 public class ValidadorEdad {
4
5 public static void validarEdad(int edad) throws EdadInvalidaException {
6 // Validaciones específicas con mensajes personalizados
7 if (edad < 0) {
8 throw new EdadInvalidaException("La edad no puede ser negativa");
9 } else if (edad > 150) {
10 throw new EdadInvalidaException("La edad parece incorrecta (muy alta)");
11 } else if (edad < 18) {
12 throw new EdadInvalidaException("Acceso denegado: Menor de edad");
13 }
14 }
15
16 public static void main(String[] args) {
17 Scanner sc = new Scanner(System.in);
18 System.out.print("Dime una edad para validar: ");
19 int edadInput = sc.nextInt();
20
21 try {
22 System.out.println("Validando edad: " + edadInput);
23 validarEdad(edadInput);
24 // Si llegamos hasta aquí, la edad es válida
25 System.out.println("Edad válida. Acceso permitido.");
26 } catch (EdadInvalidaException e) {
27 // Capturamos y manejamos nuestra excepción personalizada
28 System.out.println("Excepción capturada: " + e.getMessage());
29 } catch (Exception e) {
30 // Capturamos cualquier otra excepción genérica
31 System.out.println("Error inesperado: " + e.getMessage());
32 }
33 }
34 //Clase interna, ni public, ni static, ni private
35 class EdadInvalidaException extends Exception {
36 // Constructor que recibe un mensaje personalizado
37 public EdadInvalidaException(String s) {
38 //Llamamos al constructor de la clase base Exception y pasamos el mensaje que recibimos
39 super(s);
40 }
41 // Constructor por defecto
42 public EdadInvalidaException() {
43 super("Edad inválida");
44 }
45 }
}

```

## 9.9. Píldoras informáticas relacionadas

Videos de Makigas al respecto:

- [Java: introducción a las excepciones](#)
- [Java: throw y throws, usos y diferencias](#)

⌚ 8 de enero de 2026

## 10. 4.2 Ejercicios de la UD03

### 10.1. Retos

1. (Descuento) modifique el programa para que, en lugar de realizar un descuento del 8% si la compra es de 100 € o más, aplique una penalización de 2 € si el precio es inferior a 30 €.

```

1 import java.util.Scanner;
2 //Un programa que calcula descuentos.
3
4 public class _1_Descuento{
5 public static final float DESCUENTO= 8;
6 public static final float COMPRA_MIN = 100;
7
8 public static void main(String[] args) {
9 Scanner lector = new Scanner(System.in);
10 System.out.print("¿Cuál es el precio del producto, en euros?");
11 float precio= lector.nextFloat();
12 lector.nextLine();
13 if (precio<= COMPRA_MIN) {
14 float descuentoHecho= precio * DESCUENTO / 100;
15 precio = precio - descuentoHecho;
16 }
17 System.out.println("El precio final a pagar es de "+ precio +" euros.");
18 }
19 }
```

2. (Adivina) modifique el programa para que, en lugar de un único valor secreto, haya dos. Para ganar, basta con acertar uno de los dos. La condición lógica que necesitará ya no se puede resolver con una expresión compuesta por una única comparación. Será más compleja.

#### Acción

Para pasar satisfactoriamente los tests, la variable `VALOR_SECRETO` debe ser renombrada a `VALOR_SECRET01`, y la nueva debe llamarse `VALOR_SECRET02` (deberá generarse aleatoriamente) y aunque el nombre sea de constante, no puede ser final.

```

1 import java.util.Scanner;
2
3 public class Adivina{
4
5 public static final int VALOR_SECRETO = 4;
6
7 public static void main(String[] args) {
8 Scanner lector = new Scanner(System.in);
9 System.out.println("Empecemos el juego.");
10 System.out.print("Adivina el valor entero, entre 0 y 10: ");
11 int valorUsuario = lector.nextInt();
12 lector.nextLine();
13 if (VALOR_SECRETO == valorUsuario) {
14 System.out.println("¡Exactamente! Era " + VALOR_SECRETO + ".");
15 } else {
16 System.out.println("¡Te has equivocado!");
17 }
18 System.out.println("Hemos terminado el juego.");
19 }
20 }
```

3. (AdivinaCorrecto) modifique el ejemplo anterior (Adivina) para que comprueben que el valor que ha introducido el usuario se encuentra dentro del rango de valores correcto (entre 0 y 10).
4. (AdivinaControlErroresEntrada) aplique el mismo tipo de control sobre los datos de la entrada del ejemplo siguiente al ejercicio del reto 1.

#### Acción

Para pasar satisfactoriamente los tests, el mensaje de error cuando no se introduzca un entero debe contener la palabra "ERROR"

```

1 import java.util.Scanner;
2
3 public class AdivinaControlErroresEntrada{
4
5 public static final int VALOR_SECRETO = 4;
6
7 public static void main(String[] args) {
8 Scanner lector = new Scanner(System.in);
9 System.out.println("Empecemos el juego.");
10 System.out.print("Adivina el valor entero, entre 0 y 10: ");
11 boolean tipoCorrecto = lector.hasNextInt();
12 if (tipoCorrecto) {
13 //Se ha escrito un entero correctamente. Ya puede leerse.
14 int valorUsuario = lector.nextInt();
15 lector.nextLine();
16 if (VALOR_SECRETO == valorUsuario) {
17 System.out.println("Exacto! Era " + VALOR_SECRETO + ".");
18 } else {
19 System.out.println("Te has equivocado!");
20 }
21 System.out.println("Hemos terminado el juego.");
22 } else {
23 //No se ha escrito un entero.
24 System.out.println("El valor introducido no es un entero.");
25 }
26 }
27 }
```

5. (Linea) Modifique el ejemplo para que primero pregunte al usuario cuántos caracteres "-" quiere escribir por pantalla, y entonces los escriba. Cuando pruebe el programa, no introduzca un número muy alto!

```

1 //Un programa que escribe una linea con 100 caracteres '-'.
2
3 public class Linea {
4
5 public static void main(String[] args) {
6 //Inicializamos un contador
7
8 int i = 0;
9 //¿Ya hemos hecho esto 100 veces?
10 while (i < 100) {
11 System.out.print("-");
12 //Lo hemos hecho una vez, sumamos 1 al contador
13
14 i = i + 1;
15 }
16 //Forzamos un salto de linea
17 System.out.println();
18 }
19 }
```

6. (TablaMultiplicar) un contador tanto puede empezar a contar desde 0 e ir subiendo, como desde el final e ir disminuyendo como una cuenta atrás. Modifique este programa para que la tabla de multiplicar comience mostrando el valor para 10 y vaya bajando hasta el 1.

```

1 import java.util.Scanner;
2
3 public class TablaMultiplicar{
4
5 public static void main(String[] args) {
6 Scanner lector = new Scanner(System.in);
7 System.out.print("¿Qué tabla de multiplicar quieres? ");
8 int tabla = lector.nextInt();
9 lector.nextLine();
10 int i = 1;
11 while (i <= 10) {
12 int resultado = tabla * i;
13 System.out.println(tabla + " * " + i + " = " + resultado);
14 i = i + 1;
15 }
16 System.out.println("Ésta ha sido la tabla del " + tabla);
17 }
18 }
```

7. (Modulo) el uso de contadores y acumuladores no es excluyente, sino que puede ser complementario. Piense cómo se podría modificar el programa para calcular el resultado del módulo y la división entera a la vez. Recuerde que la división entera simplemente sería contar cuántas veces se ha podido restar el divisor.

```

1 import java.util.Scanner;
2
3 public class Modulo{
4
5 public static void main(String[] args) {
6 Scanner lector = new Scanner(System.in);
7 System.out.print("¿Cuál es el dividendo? ");
8 int dividendo = lector.nextInt();
9 lector.nextLine();
10 System.out.print("¿Cuál es el divisor? ");
11 int divisor = lector.nextInt();
12 lector.nextLine();
13 while (dividendo >= divisor) {
14 dividendo = dividendo - divisor;
15 System.out.println("Bucle: por ahora el dividendo vale " + dividendo + ".");
16 }
17 System.out.println("El resultado final es" + dividendo + ".");
18 }
19 }
```

## 10.2. Ejercicios

### 10.2.1. if else

1. (MenorDeDos) Escribir un método (`menorDeDos`) que devuelva el menor de dos números enteros introducidos por teclado.
2. (MenorDeTres) Escribir dos métodos que devuelven el menor de tres números recibidos por parámetros. Haz dos versiones:
  - `menorDeTresV1`: utilizando los operadores de comparación (`<`, `>`, `==`) y lógicos (`&&`, `||`, ... ) necesarios
  - `menorDeTresV2`: sin utilizar ninguno de los operadores lógicos (habrá que usar sentencias `if` `else` anidadas)
3. (IntermedioDeTres) Escribir un método (`intermedioDeTres`) que devuelva el intermedio de tres números recibidos por parámetros.
4. (NotasTexto) Escribir un método (`notas2Texto`) que recibe un valor numérico (se supone entre 1 y 10 y puede contener decimales) y devuelva el literal correspondiente a dicha nota según ("insuficiente", "suficiente", "bien", "notable", "sobresaliente" y "matrícula de honor").
5. (Division) Escribir un método (`division`) que reciba dos números enteros e imprima el resultado de la división. Tener en cuenta que si dividimos un número por cero se producirá un error de ejecución y debemos evitarlo. Ejemplos de ejecución:

```

1 Vamos a dividir dos números.
2 Introduce el valor del dividendo: 15
3 Introduce el valor del divisor: 3
4 La división es: 5.0
5
6 Vamos a dividir dos números.
7 Introduce el valor del dividendo: 4
8 Introduce el valor del divisor: 0
9 No se puede realizar la división porque el divisor es 0
```

6. (Raiz) Se desea calcular la raíz cuadrada real de un número real cualquiera pedido inicialmente al usuario. Como dicha operación no está definida para los números negativos es necesario tratar, de algún modo, dicho posible error sin que el programa detenga su ejecución. Debes escribir un método `raiz` que recibe un entero. Ejemplos de ejecución:

```

1 Vamos a calcular la raíz cuadrada de un número.
2 Introduce el valor del número: 81
3 La raíz es: 9.0
4
5 Vamos a calcular la raíz cuadrada de un número.
6 Introduce el valor del número: -52
7 No se puede realizar la raíz porque el número es negativo
```

7. (Hora12) Escribir un método (`formatoDoce`) que recibe dos enteros (hora y minutos) en notación de 24 horas y la devuelve como `String` en notación de 12 horas. Por ejemplo, si la entrada es 13 horas 45 minutos, la salida será "1:45 PM". La hora y los minutos se leerán de teclado de forma separada, primero la hora y luego los minutos. Si la hora pasada al método no es correcta devolverá: "Error de conversión"

```

1 Vamos a convertir una hora a formato 12H.
2 Introduce el valor para la hora: 51
3 Introduce el valor para los minutos: 6
4 Error de conversión.
5
6 Vamos a convertir una hora a formato 12H.
7 Introduce el valor para la hora: 13
8 Introduce el valor para los minutos: 45
9 01:45 PM
10
11 Vamos a convertir una hora a formato 12H.
12 Introduce el valor para la hora: 4
13 Introduce el valor para los minutos: 53
14 04:53 AM

```

8. (Bisiesto) Escribir un método (`esBisiesto`) que devuelva si un año introducido por teclado es o no bisiesto. Un año es bisiesto si es múltiplo de 4 (por ejemplo 1984). Sin embargo, los años múltiples de 100 no son bisiestos, salvo que sean múltiplos de 400, en cuyo caso si lo son (por ejemplo 1800 no es bisiesto y 2000 si lo es). Para hacer el programa, implementa un método dentro de la clase que reciba un año y devuelva true si el año es bisiesto y false en caso de que no lo sea.

9. (Fechas) Escribir un método (`imprimeMenor`) que reciba dos fechas (día, mes y año) (6 parámetros en total), que se suponen correctas, y le muestre la menor de ellas. La fecha se mostrará en formato dd/mm/año. Utiliza un método `mostrarFecha`, para mostrar la fecha por pantalla. La fecha se mostrará siempre con dos dígitos para el día, dos para el mes y cuatro para el año.

```

1 Vamos a ver que fecha es menor.
2 Introduce el valor para el Fecha1.
3 - Dia: 4
4 - Mes: 5
5 - Año: 1999
6
7 Introduce el valor para el Fecha2.
8 - Dia: 3
9 - Mes: 5
10 - Año: 1999
11
12 La fecha menor es:
13 03/05/1999

```

10. (DiasDelMes) Escribir un método (`diasMes`) que recibe el número de un mes (1 a 12) y devuelve el número de días que tiene el mes. Hacerlo utilizando sentencias `if else`. Si el mes recibido no es correcto, debe devolver 0.

```

1 Vamos a ver cuantos días tiene un mes.
2 Introduce el valor para el mes: 8
3 Este mes tiene 31 días.
4
5 Vamos a ver cuantos días tiene un mes.
6 Introduce el valor para el mes: 54
7 Este mes tiene 0 días.

```

11. (NombreDelMes) Escribir un programa que lea de teclado el número de un mes (1 a 12) y visualice el nombre del mes (enero, febrero, etc). Hacerlo utilizando sentencias `if else`. Para hacer un programa, implementa un método en la clase que reciba un número de mes y devuelva el nombre del mes

12. (Salario) Escribir un método (`salario`) que recibe las horas trabajadas por un empleado en una semana y calcula y devuelve su salario neto semanal, sabiendo que:

- Las horas ordinarias se pagan a 6 €.
- Las horas extraordinarias se pagan a 10 €.
- Los impuestos a deducir son:
  - Un 2 % si el salario bruto semanal es menor o igual a 350 €
  - Un 10 % si el salario bruto semanal es superior a 350 €
  - La jornada semanal ordinaria son 40 horas. El resto de horas trabajadas se considerarán horas extra.

13. (Signo) Dados dos números enteros, num1 y num2, realizar un programa que escriba uno de los dos mensajes:

- "el producto de los dos números es positivo o nulo" o bien
- "el producto de los dos números es negativo".

Resolverlo sin calcular el producto, sino teniendo en cuenta únicamente el signo de los números a multiplicar.

14. (Calculadora) Escribir un programa para simular una calculadora. Considera que los cálculos posibles son del tipo num1 operado num2, donde num1 y num2 son dos números reales cualesquiera y operador es una de entre: +, -, \* y /. El programa pedirá al usuario en primer lugar el valor num1, a continuación el operador y finalmente el valor num2. Resolver utilizando instrucciones `if else`

15. (Comercio) Un comercio aplica un descuento del 8% por compras superiores a 40 euros. El descuento máximo será de 12 euros. Escribir un programa que solicite al usuario el importe de la compra y muestre un mensaje similar al siguiente:

- Importe de la compra 100 €
- Porcentaje de descuento aplicado: 8%
- Descuento aplicado: 8 €
- Cantidad a pagar: 92 €

16. (Editorial) Una compañía editorial dispone de 2 tipos de publicaciones: libros y revistas. El precio de cada pedido depende del número de elementos solicitados al cual se le aplica un determinado descuento, que es diferente para libros y para revistas. La siguiente tabla muestra los descuentos a aplicar en función del número de unidades y del tipo de producto:

| Cantidad pedida         | Libros            | Revistas          |
|-------------------------|-------------------|-------------------|
| Hasta 5 unidades        | 0 % de descuento  | 0 % de descuento  |
| De 6 a 10 unidades      | 10 % de descuento | 15 % de descuento |
| De 11 a 20 unidades     | 15 % de descuento | 20 % de descuento |
| A partir de 20 unidades | 20 % de descuento | 25 % de descuento |

Escribe un método `calcularCoste` que, recibiendo el tipo de publicación (`String`), que puede ser "libro" o "revista", el precio individual (`double`) y el número de unidades solicitado (`int`), devuelva el coste del pedido (aplicando el descuento correspondiente). Escribe un programa en el que el usuario deba introducir cantidad y precio de revistas y cantidad y precio de libros que incluye un pedido, y muestre el coste del pedido.

Si el tipo especificado es diferente de "libro" o "revista", el método devolverá -1.

```

1 Vamos a calcular el coste del pedido.
2
3 Introduce los datos respecto a LIBROS.
4 Precio unitario: 12,3
5 Cantidad: 4
6
7 Introduce los datos respecto a REVISTAS.
8 Precio unitario: 2,6
9 Cantidad: 5
10
11 El coste total de pedido será de 62,20€.

```

17. (Taxi) Se desea calcular el coste del trayecto realizado en taxi en función de los kilómetros recorridos en las carreras metropolitanas de Valencia. Según las tarifas vigentes para el 2012, el coste se calcula de la siguiente manera:

- Días laborables en horario diurno (de 6:00 a antes de las 22:00h): 0.73 €/km.
- Días laborables en horario nocturno: 0.84 €/km.
- Sábados y domingos: 0.93 €/km.
- Además, la tarifa mínima diurna es de 2.95€ y la mínima nocturna de 4€.

Escribir un método que recibe:

- La hora (`hora` y `minutos`) en que se realizó el trayecto.
- El día de la semana (se supone que el usuario introduce un valor entre 1 para lunes y 7 para domingo)
- Los kilómetros recorridos.

El método devuelve el coste del trayecto. Escribe un método `main` que solicite al usuario la introducción de los datos y llamando al método muestre el resultado:

```

1 Vamos a calcular el coste del trayecto.
2 Introduce la hora: 15
3 Introduce los minutos: 20
4 Introduce el dia de la semana (1 para lunes y 7 para domingo): 6
5 Introduce los kilometros recorridos: 3,5
6
7 El coste total del trayecto es de 3,26€.

```

18. (Nombre) Escribir un método (`coincidenPrimeraYUltima`) que recibe una cadena de texto. El método devolverá si la primera y la última letra del nombre coinciden o no.

Haz que funcione aunque las letras tengan diferente CASE (pista: lowercase i uppercase).

Escribe el método `main` que solicite al usuario un nombre de persona y llamando al método diga si coinciden o no, pruébalo con "Ana", "ana", "Angel", "Amanda" y "David":

```

1 Vamos a ver si la primera y la última letra de tu nombre son iguales.
2 Introduce un nombre: Ana
3 Si coinciden.
4
5 Vamos a ver si la primera y la última letra de tu nombre son iguales.
6 Introduce un nombre: Angel
7 No coinciden.

```

19. (Validar) Se desea implementar un programa que determine si dos datos `x` e `y` de entrada son válidos. Un par de datos es válido si es uno de los que aparecen en la siguiente tabla:

| <b>x :</b>       | <b>a</b> | <b>a</b> | <b>a</b> | <b>a</b> | <b>a</b> | <b>b</b> |
|------------------|----------|----------|----------|----------|----------|----------|
| <code>y :</code> | 1        | 3        | 5        | 7        | 9        | 2        |

Se pide implementar un método (`esValido`) que recibe los dos valores (`x` e `y`), que devuelva si la combinación es válida o no.

Escribe también un método `main` que lea de teclado el valor de `x` y el valor de `y`, e indique por pantalla "VALIDOS" o "NO VALIDOS". Se pide hacerlo de forma que no se utilice ninguna estructura condicional (if, switch,...), es decir, se calculará una expresión booleana que determine si `x` e `y` son válidos. Se procurará que la expresión booleana propuesta sea breve y concisa.

### 10.2.2. Bucles simples

1. (SencillosWhile) Crear una clase llamada `SencillosWhile` y crear en él métodos que realicen las siguientes tareas.
- (imparesHastaN) Recibe un nº entero `n`, devuelve los números impares que hay entre 1 y `n`. Por ejemplo, si `n` es 8 devolverá `"1 3 5 7 "`
  - (nImpares) Recibe un nº entero `n`, devuelve los `n` primeros números impares. Por ejemplo, si `n` es 3 devolverá `"1 3 5 "` (3 primeros impares)
  - (cuentaAtras) Recibe un entero `n`, devuelve una cuenta atrás partiendo de `n`: `n, n-1, ...`. Por ejemplo, si `n` es 5 devolverá `"5 4 3 2 1 0 "`
  - (sumaNPrimeros) Recibe un entero `n`, devuelve la suma de los números entre 1 y `n`. Por ejemplo, si `n` es 5 devolverá `15`
  - (mostrarDivisoresN) Recibe un entero `n`, devuelve todos sus divisores, incluidos el 1 y el mismo `n`. Por ejemplo, si `n` es 12 mostraría `"1 2 3 4 6 12 "`
  - (sumaDivisoresN) Recibe un entero `n`, devuelve la suma de todos sus divisores, sin incluir al propio `n`. Por ejemplo, si `n` es 12 sumará `1, 2, 3, 4 y 6 = 16`

Ejemplo de ejecución con todos los métodos:

```

1 Esta realiza varios While sencillos:
2 Introduce un número: 15
3 Imprimimos:
4 Todos los números impares menores que 15: 1 3 5 7 9 11 13 15
5 Los primeros 15 números impares: 1 3 5 7 9 11 13 15 17 19 21 23 25 27 29
6 Cuenta atrás desde 15: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
7 La suma de los 15 primeros números resulta: 120
8 Los divisores del número 15 son: 1 3 5 15
9 La suma de todos los divisores del número 15 (sin él mismo) son: 9

```

2. (SencillosFor) Crear una clase llamada "SencillosFor" y crear en él los mismos métodos que en el ejercicio anterior, pero utilizando la sentencia `for` en lugar de `while`:

```

1 Esta realiza varios For sencillos:
2 Introduce un número: 15
3 Imprimimos:
4 Todos los números impares menores que 15: 1 3 5 7 9 11 13 15
5 Los primeros 15 números impares: 1 3 5 7 9 11 13 15 17 19 21 23 25 27 29
6 Cuenta atrás desde 15: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
7 La suma de los 15 primeros números resulta: 120
8 Los divisores del número 15 son: 1 3 5 15
9 La suma de todos los divisores del número 15 (sin él mismo) son: 9

```

3. (PotenciasDe2) Crea un método (`potencias`) que recibe un entero `n` positivo y devuelve las `n` primeras potencias de 2. Es decir,  $2^0, 2^1, 2^2, 2^3, \dots, 2^n$ . Crea un método `main` que solicite un número al usuario y muestre por pantalla todas las potencias de 2. Soluciona el ejercicio sin utilizar `Math.pow`. Ten en cuenta que, por ejemplo,  $2^3 = 1 * 2 * 2 * 2$  o que  $2^4 = 1 * 2 * 2 * 2 * 2$ :

```

1 Dime n: 20
2 1 2 4 8 16 32 64 128 256 512 1024 2048 4096 8192 16384 32768 65536 131072 262144 524288 1048576

```

4. (Etapas) El ser humano pasa por una serie de etapas en su vida que, con carácter general se asocian a las edades que aparecen en la tabla siguiente.

| Etapa        | Rango                   |
|--------------|-------------------------|
| Infancia     | Hasta los 10 años       |
| Pubertad     | De 11 a 14 años         |
| Adolescencia | De 15 a 21 años         |
| Adulvez      | De 22 a 54 años         |
| Vejez        | De 55 a 70 años         |
| Ancianidad   | A partir de los 71 años |

Escribe un método `main` en el que el usuario introduzca las edades de una serie de personas y calcule y muestre que porcentaje de personas que se encuentran en cada etapa. En primer lugar el programa pedirá el número de personas que participan en la muestra y a continuación solicitará la edad de cada una de ellas. El resultado será similar al siguiente:

```

1 ¿Cuántas personas hay en la muestra?: 5
2 Introduce la edad de la persona 1 de 5 : 17
3 --> adolescencia
4 Introduce la edad de la persona 2 de 5 : 21
5 --> adolescencia
6 Introduce la edad de la persona 3 de 5 : 53
7 --> adulvez
8 Introduce la edad de la persona 4 de 5 : 62
9 --> vejez
10 Introduce la edad de la persona 5 de 5 : 78
11 --> ancianidad
12 Etapa Número %
13 Infancia: 0 0.0%
14 Pubertad: 0 0.0%
15 Adolescencia: 2 40.0%
16 Adulvez: 1 20.0%
17 Vejez: 1 20.0%
18 Ancianidad: 1 20.0%
19
20 ¿Cuántas personas hay en la muestra?: 3
21 Introduce la edad de la persona 1 de 3 : 1
22 --> infancia
23 Introduce la edad de la persona 2 de 3 : 2
24 --> infancia
25 Introduce la edad de la persona 3 de 3 : 25
26 --> adulvez
27 Etapa Número %
28 Infancia: 2 66.66666666666667%
29 Pubertad: 0 0.0%
30 Adolescencia: 0 0.0%
31 Adulvez: 1 33.33333333333336%
32 Vejez: 0 0.0%
33 Ancianidad: 0 0.0%
```

5. (Primo) Escribir un método `esPrimo` que recibe un entero y devuelve si es primo o no. Escribe un método `main` en el que el usuario escriba un número entero y se le diga si se trata o no de un número primo, usando el método `esPrimo`. Recuerda que un nº primo es aquel que solo es divisible por 1 y por sí mismo (Es decir tiene SOLO y EXCLUSIVAMENTE dos divisores cuyo resto sea cero).

```

1 Este te dice si un número es primo:
2 Introduce un número: 15
3 El número es primo?: false
4
5 Este te dice si un número es primo:
6 Introduce un número: 7
7 El número es primo?: true
```

6. (Primos) Escribir un programa en el que el usuario escriba un número entero y se le diga todos los números primos entre 1 y el número introducido (Puedes usar el método `esPrimo` que has escrito en el ejercicio anterior).

7. (EsPrimoMejorada) Haz una nueva versión del programa del ejercicio anterior teniendo en cuenta lo siguiente:

- El único número par que es primo es el 2.
- Un número  $n$  no puede tener divisores mayores que  $n/2$  (o mayores que `Math.sqrt(n)`)

**A**tenCIÓN

Para pasar satisfactoriamente los tests de rendimiento, debes tener el ejercicio 6 (Primos) y 7 (EsPrimoMejorada) y el rendimiento del 7 debe ser un 20% mejor (más rápido) que el del 6

8. (Divisores) Escribir un programa que muestre los tres primeros divisores de un número n introducido por el usuario. Por ejemplo, si el usuario introduce el número 45, el programa mostrará los divisores 1, 3 y 5. Ten en cuenta que la posibilidad de que el número n tenga menos de 3 divisores. Prueba qué pasa si el usuario pide, por ejemplo, los tres primeros divisores de 7.
9. (SumaSerie) Dado un número n, introducido por el usuario, calcula y muestra por pantalla la siguiente suma  $1/1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}$
10. (Cifras) Escribir un método (`numeroCifras`) que recibe un número entero cualquiera (positivo, negativo o cero) y devuelve cuantas cifras tiene. Pistas: ¿Cuántas cifras tiene el nº 25688? ¿Cuántas veces podemos dividir el nº 25688 por 10 hasta que se hace cero? Cuidado, el nº 0 tiene una cifra.

```

1 Vamos a decirte cuantas cifras tiene un número:
2 Introduce un número: 0
3 El número tiene 1 cifras.
4
5 Vamos a decirte cuantas cifras tiene un número:
6 Introduce un número: 25688
7 El número tiene 5 cifras.
8
9 Vamos a decirte cuantas cifras tiene un número:
10 Introduce un número: -14
11 El número tiene 2 cifras.

```

11. (Transportes) Una empresa de transportes cobra 30€ por cada bulto que transporta. Además, si el peso total de todos los bultos supera los 300 kilos, cobra 0.9€ por cada kg extra. Por último si el trasporte debe realizarse en sábado, cobra un plus de 60€. La empresa no realiza el pedido si hay que transportar más de 30 bultos, si el peso total supera los 1000 kg o si se solicita hacerlo en domingo. Realizar un método `main` que solicite el número de bultos, el día de la semana (valor entre 1 y 7) y el peso de cada uno de los bultos y muestre el coste del transporte en caso de que pueda realizarse o un mensaje adecuado en caso contrario:

```

1 ¿Cuantos bultos debes enviar?: 3
2 ¿Qué dia de la semana se realiza el envio (entre 1=lunes y 7=domingo)?: 7
3 No se puede realizar el envio por superar los 30 bultos o ser domingo
4
5 ¿Cuantos bultos debes enviar?: 4
6 ¿Qué dia de la semana se realiza el envio (entre 1=lunes y 7=domingo)?: 6
7 Cuanto pesa el bulto 1 de 4 : 25
8 Cuanto pesa el bulto 2 de 4 : 120
9 Cuanto pesa el bulto 3 de 4 : 35
10 Cuanto pesa el bulto 4 de 4 : 78
11 El coste del envio será de: 180.0
12
13 ¿Cuantos bultos debes enviar?: 2
14 ¿Qué dia de la semana se realiza el envio (entre 1=lunes y 7=domingo)?: 1
15 Cuanto pesa el bulto 1 de 2 : 700
16 Cuanto pesa el bulto 2 de 2 : 400
17 No se puede realizar el envio por superar los 1000Kg

```

12. (Containers) La capacidad de un buque que transporta containers está limitada tanto por la cantidad de containers como por el peso, pudiendo transportar un máximo de 100 containers y un máximo de 700 toneladas. Hacer un programa en el que se vaya introduciendo el peso de los containers (en toneladas) a medida que se cargan en el barco, hasta que se llegue al máximo de capacidad. Mostrar al final la cantidad de containers cargados y el peso total. En el momento en que se desee cargar un container que haga que la carga total supere las 700 toneladas, se dará por finalizada la carga, aunque pudieran existir containers menos pesados con posibilidad de ser cargados.
13. (Notas) Realizar un método `main` que permita introducir las notas de varios exámenes de un alumno de un curso. El usuario irá introduciendo las notas una tras otra. Se considerará finalizado el proceso de introducción de notas cuando el usuario introduzca una nota negativa. Al final, el programa le dirá: - El número de notas introducidas. - El número de aprobados (mayor o igual a 5 puntos) - La nota media

```

1 Vamos a darte información sobre tus notas, introduce una nota tras otra. Para finalizar escribe una nota negativa:
2 Introduce la nota 1: 5,5
3 Introduce la nota 2: 6,7
4 Introduce la nota 3: 8,4
5 Introduce la nota 4: 3,2
6 Introduce la nota 5: 1,5
7 Introduce la nota 6: -3
8 Has introducido un total de 5 notas,
9 de las cuales has aprobado 3,
10 con una media de 4.

```

14. (NotasExtremas) Modificar el ejercicio anterior para que además calcule la nota máxima y la nota mínima.

### 10.2.3. Bucles anidados

1. (Edades) Programa que pida al usuario la edad de cinco personas. Si la suma de las edades es inferior a 200, el programa volverá a solicitar las 5 edades.

```

1 Introduce la edad de la persona 1: 80
2 Introduce la edad de la persona 2: 42
3 Introduce la edad de la persona 3: 15
4 Introduce la edad de la persona 4: 56
5 Introduce la edad de la persona 5: 34

```

```

1 Introduce la edad de la persona 1: 15
2 Introduce la edad de la persona 2: 12
3 Introduce la edad de la persona 3: 43
4 Introduce la edad de la persona 4: 5
5 Introduce la edad de la persona 5: 4
6 La suma de edades debe ser superior a 200
7
8 Introduce la edad de la persona 1: 48
9 Introduce la edad de la persona 2: 54
10 Introduce la edad de la persona 3: 35
11 Introduce la edad de la persona 4: 12
12 Introduce la edad de la persona 5: 89

```

2. (NotasPorAlumno) Programa que pida al usuario las notas de `A` alumnos en `s` asignaturas, alumno por alumno. `A` y `s` se definirán en el programa como `CONSTANTES`.

#### Atención

Para pasar satisfactoriamente los tests, la variable `A` debe ser 3 y `s` debe ser 5.

```

1 Alumno 1
2 Introduce la nota de la asignatura 1: 5
3 Introduce la nota de la asignatura 2: 4
4 Introduce la nota de la asignatura 3: 3
5 Introduce la nota de la asignatura 4: 5
6 Introduce la nota de la asignatura 5: 0
7 Alumno 2
8 Introduce la nota de la asignatura 1: 6
9 Introduce la nota de la asignatura 2: 8
10 Introduce la nota de la asignatura 3: 9
11 Introduce la nota de la asignatura 4: 8
12 Introduce la nota de la asignatura 5: 7
13 Alumno 3
14 Introduce la nota de la asignatura 1: 0
15 Introduce la nota de la asignatura 2: 0
16 Introduce la nota de la asignatura 3: 2
17 Introduce la nota de la asignatura 4: 4
18 Introduce la nota de la asignatura 5: 5

```

3. (NotasPorAsignatura) Programa que pida al usuario las notas de `A` alumnos en `s` asignaturas, asignatura por asignatura. `A` y `s` se definirán en el programa como `CONSTANTES`.

```

1 Asignatura 1
2 Introduce nota del alumno 1:
3 Introduce nota del alumno 2:
4 ...
5 Asignatura 2
6 Introduce nota del alumno 1:
7 ...

```

4. (MediasPorAsignatura) Repite el ejercicio anterior haciendo que se muestre la media de cada asignatura

```

1 Asignatura 1
2 Introduce nota del alumno 1:
3 Introduce nota del alumno 2:
4 ...
5 Media asignatura 1: 8.5 puntos
6
7 Asignatura 2
8 Introduce nota del alumno 1:
9 ...
10 Media asignatura 2: 6.5 puntos
11 ...

```

5. (TablaMult) Escribir un programa que permita al usuario introducir un número `N` e imprima la tabla de multiplicar (del 0 al 10) de todos los números entre 1 y `N`. Ejemplo: Si el usuario introduce en número 5, el programa imprimiría

```

1 Tabla del 1:
2 1 por 0, 0
3 1 por 1, 1
4 1 por 2, 2
5 ...
6 1 por 10, 10
7
8 Tabla del 2:
9 2 por 0, 0
10 2 por 1, 2
11 ...
12 2 por 10, 20
13
14 Tabla del 3:
15 ...
16
17 Tabla del 5:
18 ...
19 5 por 10, 50

```

6. (PrimosHastaN) Programa que solicite al usuario un numero `n` y muestre todos los números primos menores o iguales que `n`. (IGUAL AL 27!!)

7. (CombinarLetras2) Escribir un programa que muestre todas las palabras de dos letras que se pueden formar con los cuatro primeros caracteres del alfabeto en minúsculas ('a', 'b', 'c', 'd'):

```

1 aa
2 ab
3 ac
4 ad
5 ba
6 bb
7 bc
8 bd
9 ...
10 da
11 db
12 dc
13 dd

```

### Acción

Para pasar satisfactoriamente los tests, en lugar de usar `System.out.println`, debes usar `System.out.print` con un `\n` al final de la cadena a imprimir.

8. (CombinarLetras3) Repite el ejercicio anterior mostrando palabras de tres letras

```

1 aaa
2 aab
3 ...
4 ddc
5 ddd

```

### Acción

Para pasar satisfactoriamente los tests, en lugar de usar `System.out.println`, debes usar `System.out.print` con un `\n` al final de la cadena a imprimir.

9. (LetraALetra) Escribe un programa en el que se solicite al usuario un texto de forma repetida hasta que el usuario introduzca la cadena vacía. Con cada texto que introduzca el usuario se le mostrará carácter a carácter, cada carácter en una línea

```

1 Vamos a escribir textos letra a letra
2 Introduce texto: Hola
3 H
4 o
5 l
6 a
7 Introduce texto: Casa
8 C
9 a
10 s
11 a
12 Introduce texto:
13 Fin del programa

```

### Acción

Para pasar satisfactoriamente los tests, en lugar de usar `System.out.println`, debes usar `System.out.print` con un `\n` al final de la cadena a imprimir.

10. (DibujarFiguras1) Escribe una clase que contenga los métodos que se indican a continuación. En el método `main` solicita al usuario las dimensiones de las figuras necesarias en cada caso y llama al método correspondiente para que se muestre por pantalla
- a. `void dibRecAsteriscos (int ancho, int alto)` dibuja un rectángulo utilizando asteriscos, como el siguiente. En el ejemplo ancho es 7 y alto es 3

```

1 * * * * * *
2 * * * * * *
3 * * * * * *

```

```

1 Vamos a dibujar figuras.
2 Introduce el ancho: 8
3 Introduce el alto: 4
4 * * * * * *
5 * * * * * *
6 * * * * * *
7 * * * * * *

```

- b. `void dibRecNumeros1 (int ancho, int alto)` dibuja un rectángulo utilizando números, como el siguiente. En el ejemplo ancho es 7 y alto es 3

```

1 1 2 3 4 5 6 7
2 1 2 3 4 5 6 7
3 1 2 3 4 5 6 7

```

- c. `void dibRecNumeros2 (int ancho, int alto)` dibuja un rectángulo utilizando números, como el siguiente. En el ejemplo ancho es 7 y alto es 3

```

1 7 6 5 4 3 2 1
2 7 6 5 4 3 2 1
3 7 6 5 4 3 2 1

```

- d. `void dibRecNumeros3 (int ancho, int alto)` dibuja un rectángulo utilizando números, como el siguiente. En el ejemplo ancho es 7 y alto es 3

```

1 01 02 03 04 05 06 07
2 08 09 10 11 12 13 14
3 15 16 17 18 19 20 21

```

- e. `void dibDiagonal (int ancho, int alto)` dibuja un rectángulo con ceros y unos. Los 1 están en las posiciones en las que fila y columna coinciden. En el ejemplo ancho es 7 y alto es 3

```

1 1 0 0 0 0 0 0
2 0 1 0 0 0 0 0
3 0 0 1 0 0 0 0

```

- f. `void dibRecLetras (int ancho, int alto)` dibuja un rectángulo letras sucesivas comenzando por la "a". En el ejemplo ancho es 7 y alto es 3

```

1 a a a a a a a
2 b b b b b b b
3 c c c c c c c

```

g. void dibRecLetras2 (int ancho, int alto) dibuja un rectángulo letras sucesivas terminando por la "a". En el ejemplo ancho es 7 y alto es 3

```

1 c c c c c c c
2 b b b b b b b
3 a a a a a a a

```

h. void dibRecLetras3 (int ancho, int alto) dibuja un rectángulo letras sucesivas comenzando por la "a". En el ejemplo ancho es 7 y alto es 3

```

1 a b c d e f g
2 h i j k l m n
3 o p q r s t u

```

11. (dibujarFiguras2) Escribe una clase que contenga los métodos que se indican a continuación. En el método main solicita al usuario las dimensiones de las figuras necesarias en cada caso y llama al método correspondiente para que se muestre por pantalla

a. void dibRectNumeros3 (int ancho, int alto) dibuja un rectángulo utilizando números, como el siguiente. En el ejemplo ancho es 7 y alto es 3

```

1 1 2 3 4 5 6 7 7 6 5 4 3 2 1
2 1 2 3 4 5 6 7 7 6 5 4 3 2 1
3 1 2 3 4 5 6 7 7 6 5 4 3 2 1

```

b. void dibRectAsteriscos1 (int ancho, int alto) dibuja un rectángulo utilizando asteriscos (\*) y espacios en blanco, como el siguiente. En el ejemplo ancho es 7 y alto es 3

```

1 * * * * * *
2 * * * * * *
3 * * * * * *

```

c. void dibRectAsteriscos2 (int ancho, int alto) dibuja un rectángulo utilizando asteriscos (\*), espacios en blanco y el carácter '+', como el siguiente. En el ejemplo ancho es 7 y alto es 3

```

1 * + * + * + *
2 * + * + * + *
3 * + * + * + *

```

d. void dibRectAsteriscos3 (int ancho, int alto) dibuja un rectángulo utilizando asteriscos (\*) y espacios en blanco, como el siguiente. En el ejemplo ancho es 7 y alto es 3

```

1 * * * * * *
2 *
3 * * * * * *

```

e. void dibTriangulo1 (int base) dibuja un triángulo utilizando asteriscos (\*) y espacios en blanco, como el siguiente. En el ejemplo base es 5

```

1 *
2 **
3 ***
4 ****
5 *****

```

f. void dibTriangulo2 (int altura) dibuja un triángulo utilizando asteriscos (\*) y espacios en blanco, como el siguiente. En el ejemplo altura es 5

```

1 *
2 **
3 ***
4 ****
5 *****

```

g. void dibTriangulo3 ( int altura) dibuja un triángulo utilizando asteriscos (\*) y espacios en blanco, como el siguiente. En el ejemplo altura es 5

```

1 *
2 * *
3 * * *
4 * * * *
5 * * * * *

```

#### 10.2.4. switch

- (NotasTexto2) Escribir un método (`nota2TextoSwitch`) que reciba la nota de un examen (valor numérico entero entre 1 y 10) y muestre el literal correspondiente a dicha nota según (insuficiente, suficiente, bien, notable, sobresaliente, matrícula de honor). Hacerlo utilizando la sentencias switch.
- (DiasDelMes2) Escribir un método que recibe un número de mes (1 a 12) y devuelva el número de días que tiene el mes. Resolver utilizando la sentencias switch. Si el mes es erroneo, devolverá 0.

```

1 Vamos a calcular el número de días que tiene un mes.
2 Introduce el número del mes (1-12): 8
3 31
4
5 Vamos a calcular el número de días que tiene un mes.
6 Introduce el número del mes (1-12): 2
7 28
8
9 Vamos a calcular el número de días que tiene un mes.
10 Introduce el número del mes (1-12): 15
11 0

```

- (NombreDelMes2) Escribir un método (`nombreMes`) que recibe el número de un mes (1 a 12) y devuelve el nombre del mes ("enero", "febrero", etc). Resolver utilizando la sentencias switch. Si el mes no es válido, devolverá la cadena vacía "".
- (Calculadora2) Escribir un método (`calculadora`) para simular una calculadora. Considera que los cálculos posibles son del tipo `num1, operador, num2`, donde `num1` y `num2` son dos números reales cualesquiera y operador es una de entre: '+', '-', '\*' y '/'. El método recibirá los tres parámetros y devolverá el resultado de la operación. Resolver utilizando la sentencias switch. Si el `operador` no es ninguno de la lista, devuelve el `num1`.

#### 10.2.5. en papel...

- (Valor) ¿Qué valor se asignará a consumo en la sentencia `if` siguiente si velocidad es 120?

```

1 if (velocidad > 80)
2 consumo = 10;
3 else if (velocidad > 100)
4 consumo = 12;
5 else if (velocidad > 120)
6 consumo = 15;

```

- (Errores) Encuentra y corrige los errores de los siguientes fragmentos de programa.

## a. fragmento a

```

1 if (x > 25)
2 y = x
3 else
4 y = z;

```

## b. fragmento b

```

1 if (x<0)
2 System.out.println("El valor de x es" +x);
3 System.out.println ("x es negativo");
4 else
5 System.out.println ("El valor de x es"+x);
6 System.out.println ("x es positivo");

```

## c. fragmento c

```

1 if (x = 0) System.out.println ("x igual a cero");
2 else System.out.println ("x distinto de cero");

```

## 3. (SalidaExacta) Cuál es la salida exacta por pantalla del siguiente fragmento de programa

```

1 int x = 20;
2 System.out.println("Comenzamos");
3 if (x<= 20)
4 if (x>50) System.out.println("Muy grande");
5 else {
6 if (x%2 != 0) System.out.println("Impar");
7 }
8 else if (x<=20) System.out.println("Pequeño");
9 System.out.println("Terminamos");

```

## 4. (Descuentos) En una tienda, por liquidación, se aplican distintos descuentos en función del total de las compras realizadas:

- Si total < 500 €, no se aplica descuento.
- Si 500 € <= total <= 2000 €, se aplica un descuento del 30 %.
- Si total > 2000 €, entonces se aplica un descuento del 50 %

¿Cuál de los siguientes fragmentos de programa asigna a la variable `desc` el descuento correcto? Indica "Si" o "NO" al lado de cada fragmento

## a. fragmento a

```

1 double desc = 0.0;
2 if (total <= 500)
3 if (total >= 2000) desc = 30.0;
4 else desc = 50.0;
5 total = total * desc / 100.0;

```

## b. fragmento b

```

1 double desc = 0.0;
2 if (total >= 500)
3 if (total <= 2000) desc = 30.0;
4 else desc = 50.0;
5 total = total * desc / 100.0;

```

## c. fragmento c

```

1 double desc = 0.0;
2 if (total <= 2000){
3 if (total >= 500) desc = 30.0;
4 } else desc = 50.0;
5 total = total * desc / 100.0;

```

## d. fragmento d

```

1 double desc = 0.0;
2 if (total > 500)
3 if (total < 2000) desc = 30.0;
4 else desc = 50.0;
5 total = total * desc /100.0;

```

5. (Salida) ¿Qué salida producirá el siguiente fragmento de programa si la variable entera platos vale 1? ¿Y si vale 3? ¿Y si vale 0?

```

1 switch (platos) {
2 case 1: System.out.println("\nPrimer plato");
3 case 2: System.out.println ("\nSegundo plato");
4 case 3: System.out.println ("\\nBebida");
5 System.out.println ("\nPostre");
6 break;
7 default: System.out.println("\nCafé");
8 }

```

6. (ValorP) Dados tres enteros a, b y c, y un booleano p, el siguiente análisis por casos establece el valor de p en función de los valores de a, b y c:

```

1 Si a > b entonces p = cierto;
2 si a < b entonces p = falso;
3 si a = b entonces
4 si a > c entonces p = cierto;
5 si a < c entonces p = falso;
6 si a = c entonces p = falso;

```

Se pide la traducción de dicho análisis por casos a Java mediante:

- Una única instrucción `if` sin anidamientos.
- Una única instrucción, de la forma `p = ...`, que utilice el operador ternario.
- Una única instrucción, de la forma `p = ...`, sin sentencias `if` ni utilizar el operador ternario.

#### 10.2.6. Trazas

Indica cual será la salida producida por los siguientes programas, teniendo en cuenta los datos de entrada:

1. (Trazado) **Datos de entrada: 2, 5**

a.

```

1 public static void main (String[] args){
2 Scanner tec = new Scanner(System.in);
3 int x,y,a;
4 x = tec.nextInt();
5 y = tec.nextInt();
6 a = x+y;
7 System.out.println(a);
8 }

```

b.

```

1 public static void main (String[] args){
2 Scanner tec = new Scanner(System.in);
3 int x,a;
4 x = tec.nextInt();
5 x = tec.nextInt();
6 a= x+x;
7 System.out.println(a);
8 }

```

c.

```

1 public static void main (String[] args){
2 Scanner tec = new Scanner(System.in);
3 int x,y,a;
4 x = tec.nextInt();
5 y = tec.nextInt();
6 a = x+y;
7 a = x*y;
8 System.out.println(a);
9 }

```

d.

```

1 public static void main (String[] args){
2 Scanner tec = new Scanner(System.in);
3 int x,y,a;
4 x = tec.nextInt();
5 y = tec.nextInt();
6 a = x+y;
7 System.out.println(a);
8 a = x*y;
9 System.out.println(a);
10 }

```

e.

```

1 public static void main (String[] args){
2 Scanner tec = new Scanner(System.in);
3 int x,y,a;
4 x = tec.nextInt();
5 y = tec.nextInt();
6 a = x*y;
7 a = a*x+y;
8 a = a*a;
9 System.out.println(a);
10 }

```

f.

```

1 public static void main (String[] args){
2 Scanner tec = new Scanner(System.in);
3 int x,y,a;
4 x = tec.nextInt();
5 y = tec.nextInt();
6 a = x;
7 a = doble(x);
8 System.out.format ("%d\n%d\n%d",x,y,a);
9 }
10 public static int doble(int num){
11 return 2*num;
12 }

```

g.

```

1 public static void main (String[] args) {
2 Scanner tec = new Scanner(System.in);
3 int x,y,a;
4 x = tec.nextInt();
5 y = tec.nextInt();
6 a = x;
7 doble(a);
8 System.out.format("%d\n%d\n%d\n%d",x,y,a);
9 }
10 public static void doble(int x){
11 x = 2*x;
12 }

```

h.

```

1 public static void main (String[] args){
2 Scanner tec = new Scanner(System.in);
3 int x,y,a;
4 x = tec.nextInt();
5 y = tec.nextInt();
6 a = calcular(y,x);
7 System.out.format("%d\n%d\n%d\n%d",x,y,a);
8 }
9 public static int calcular (int x, int y){
10 return x-y;
11 }

```

i.

```

1 public static void main (String[] args){
2 Scanner tec = new Scanner(System.in);
3 int x,y,a;
4 x = tec.nextInt();
5 y = tec.nextInt();
6 y = calcular(x);
7 a = calcular(y);
8 System.out.format("%d\n%d\n%d\n%d",x,y,a);
9 }
10 public static int calcular (int x){
11 return x*x;
12 }

```

2. (Traza2) **Datos de entrada: 2, 5, 7**

```

1 public static void main (String[] args){
2 int k,l,m,x,y,z;
3 k = tec.nextInt();
4 l = tec.nextInt();
5 m = tec.nextInt();
6 x = k+l;
7 if (x != m) {
8 y = k*l;
9 z = 0;
10 } else {
11 y = 0;
12 z = k-l;
13 }
14 if (z < 0) z = -z;
15 System.out.format("%d\n%d\n%d\n",x,y,z);
16 }
```

3. (Traza3) **Datos de entrada: 2, 5, 7, 9, -9, -7, -5, -2**

a.

```

1 public static void main (String[] args){
2 int x,y;
3 x = 0;
4 y = tec.nextInt();
5 while(! (y<0)) {
6 x+=-y;
7 y = tec.nextInt();
8 System.out.format("%d, %d",x,y);
9 }
10 }
```

b.

```

1 public static void main (String[] args){
2 int x,y,z,a;
3 x = y = z = a = 0;
4 x = tec.nextInt();
5 while(x>0) {
6 if (y < z) y = tec.nextInt();
7 else z= tec.nextInt();
8 a = a-x+y*z;
9 x = tec.nextInt();
10 System.out.format("%d, %d, %d, %d",a,x,y,z);
11 }
12 }
```

4. (Traza4) **Datos de entrada: 5, 5, 7, -5, -4, 2**

a.

```

1 public static void main (String[] args){
2 int x, y, a=0;
3 x = 0;
4 y = 99;
5 while (x >= 0) {
6 x = tec.nextInt();
7 y = tec.nextInt();
8 a = a + x*y;
9 }
10 System.out.println(a);
11 }
```

b.

```

1 public static void main (String[] args){
2 int x, y, a=0;
3 x = 0;
4 y = 99;
5 while (x >= 0 && y >= 0) {
6 x = tec.nextInt();
7 y = tec.nextInt();
8 a = a + x*y;
9 }
10 System.out.println(a);
11 }
```

c.

```

1 public static void main (String[] args){
2 int x, y, a=0;
3 x = 0;
4 y = 99;
5 while (x >= 0 && y <= 0) {
6 x = tec.nextInt();
7 y = tec.nextInt();
8 a = a + x*y;
9 }
10 System.out.println(a);
11 }
```

d.

```

1 public static void main (String[] args){
2 int x, y, a=0;
3 x = 0;
4 y = 99;
5 while (x >= 0 || y >= 0) {
6 x = tec.nextInt();
7 y = tec.nextInt();
8 a = a + x*y;
9 }
10 System.out.println(a);
11 }
```

5. (Trazo5) **Datos de entrada: 5, 5, 7, -5, -4, 2**

```

1 public static void main(String[] args) {
2 int x, y;
3 x = 2;
4 y = 3;
5 while (x + y > 0) {
6 x = tec.nextInt();
7 y = tec.nextInt();
8 x += y;
9 y = x - y;
10 System.out.format("%d, %d", x, y);
11 }
12 }
```

6. (Trazo6) **Datos de entrada: 2, 4, 7, 5, -6, -3, 6, 6**

a.

```

1 public static void main (String[] args){
2 int a,b;
3 do{
4 a = tec.nextInt();
5 b = tec.nextInt();
6 for (int i=a ; i<=b ; i++)
7 System.out.println(i);
8 } while (a!=b)
9 }
```

b.

```

1 public static void main (String[] args){
2 int a,b;
3 a=5;
4 b=5;
5 do {
6 for (int i=a ; i<=b ; i++)
7 System.out.println(i);
8 a = tec.nextInt();
9 b = tec.nextInt();
10 } while (a!=b);
11 }
```

7. (Trazo7) **Datos de entrada: 3, 3, 5, 5, -3, -7, 2, 2**

```

1 public static void main (String[] args){
2 int x,y;
3 do {
4 x = tec.nextInt();
5 b = tec.nextInt();
6 } while (x==y);
7 if (x>y) {
8 x=y;
9 y=x;
10 }
11 System.out.format("%d %d %n",x,y);
12 }
```

**8. (Trazo8) Datos de entrada: 3, 2, 1, 4**

```

1 public static void main (String[] args){
2 int a=0,b;
3 b = tec.nextInt();
4 for(int i=1;i<=b,i++) a=(a+i)*i;
5 System.out.println(a);
6 }
```

**9. (Trazo9) Datos de entrada: No aplica**

```

1 public static void main (String[] args){
2 int x,y;
3 for (x=3;x>=1;x--){
4 for(y=1;y<=x;y++) System.out.println(x);
5 System.out.println();
6 }
7 }
```

**10. (Trazo10) Datos de entrada: No aplica**

```

1 public static void main (String[] args){
2 int x,y;
3 x=0;
4 y=0;
5 for (int i=1;i<=2;i++) {
6 for (int j=1;j<=3;j++) x=(x+i)*j;
7 y+=x;
8 }
9 System.out.println("%d %d %n",x,y);
10 }
```

**11. (Trazo11) Datos de entrada: 4, 5, 6, 7, 8, 9**

```

1 public static void main (String[] args){
2 int x,y;
3 do x = tec.nextInt();
4 while (x<=5);
5 y=0;
6 for (int i=12;i>=x;i-=2) y+=(x*i);
7 System.out.println(y);
8 }
```

### 10.2.7. Excepciones

1. (Edades) Escribe un programa que solicite al usuario la edad de cinco personas y calcule la media. La edad de una persona debe ser un valor entero comprendido en el rango [0,110]. Realiza tres versiones:

- a. (Edades\_1) Si se introduce mal la edad de una persona se vuelve a pedir la edad de esa persona.

```

1 Introduce una edad entre 0 y 110 para la persona 1: 55
2 Introduce una edad entre 0 y 110 para la persona 2: 125
3 Introduce una edad entre 0 y 110 para la persona 2: -4
4 Introduce una edad entre 0 y 110 para la persona 2: 34
5 Introduce una edad entre 0 y 110 para la persona 3: 45
6 Introduce una edad entre 0 y 110 para la persona 4: 45
7 Introduce una edad entre 0 y 110 para la persona 5: 5
8 La media de las 5 edades introducidas es: 30,20
```

- a. (Edades\_2) Si se introduce mal la edad de una persona, el programa muestra un mensaje de error, no calcula la media y termina.

```

1 Introduce una edad entre 0 y 110 para la persona 1: 55
2 Introduce una edad entre 0 y 110 para la persona 2: 125
3 java.lang.Exception: La edad introducida debe estar entre 0 y 110.

```

- a. (Edades\_3) Si se introduce mal la edad de una persona, el programa vuelve a solicitar la edad de las cinco personas (comienza el proceso).
2. (PosicionLetra) Escribe los programas que se indican a continuación. Ejecuta cada programa haciendo que la entrada del usuario provoque una excepción. Anota el nombre de la excepción que se produce y cuál es la jerarquía de objetos de la que desciende:
- Programa que solicita dos números enteros (a y b) y muestra el resultado de su división (a/b).
    - El usuario introduce 0 como valor de b.
    - El usuario introduce letras cuando el programa espera números enteros.
    - El usuario introduce un número real cuando el programa espera un entero.
  - Programa que solicita al usuario su nombre y una posición dentro del nombre. Se muestra al usuario la letra del nombre cuya posición se ha indicado. Por ejemplo:

```

1 Introduce nombre: Javi
2 Introduce posición: 2
3 En la posición 2 de Javi está la letra a

```

- El usuario introduce una posición inválida.
- (PosicionLetraMain) Repite el ejercicio anterior utilizando métodos y llamándolos desde el método `main`:

  - Un método `dividir` que devuelva el cociente de dos números que recibe como parámetro
  - Un método `letraNombre` que, dados un String `nombre` y un entero `pos`, devuelva el carácter del nombre que ocupa la posición indicada. Ejecuta los programas provocando errores (como en el ejercicio anterior) y observa los mensajes que se generan.

- (DividirArgs) Escribir un programa que divida dos números que se reciben en main en `args[0]` y `args[1]`.

Ejemplo:

```

1 $ java dividir 10 5
2 10/5 es igual a 2

```

Donde 10 y 5 son `args[0]` y `args[1]` respectivamente, es decir los parámetros con que llamamos al programa `dividir`.

5. (PorqueError) Justifica por qué se produce error en el siguiente fragmento de código

```

1 try {
2 System.out.println("Introduce edad: ");
3 int edad = tec.nextInt();
4 if (edad >= 18) {
5 System.out.println("Mayor edad");
6 } else {
7 System.out.println("Menor edad");
8 }
9 System.out.println("Introduce nif");
10 String nif = tec.next();
11 int numero = Integer.parseInt(nif.substring(0, nif.length() - 1));
12 char letra = nif.charAt(nif.length() - 1);
13 System.out.println("Número: " + numero);
14 System.out.println("Letra: " + letra);
15 } catch (Exception e){
16 System.out.println("Debias introducir un número");
17 } catch (NumberFormatException e) {
18 System.out.println("El nif es incorrecto");
19 }

```

6. (SalidaPantalla) Indica qué se mostrará por pantalla cuando se ejecute esta clase y por qué:

```

1 public class Uno {
2 private static int metodo() {
3 int valor=0;
4 try {
5 valor = valor + 1;
6 valor = valor + Integer.parseInt("42") ;
7 valor = valor + 1;
8 System.out.println("Valor al final del try: " + valor);
9 } catch(NumberFormatException e) {
10 valor = valor + Integer.parseInt ("42");
11 System.out.println("Valor al final del catch: " + valor) ;
12 }
13 finally {
14 valor = valor + 1;
15 System.out.println("Valor al final de finally: " + valor) ;
16 }
17 valor = valor + 1;
18 System.out.println ("Valor antes del return: " + valor) ;
19 return valor;
20 }
21
22 public static void main(String[] args) {
23 try {
24 System.out.println (metodo());
25 } catch (Exception e) {
26 System.err.println("Excepcion en metodo()");
27 e.printStackTrace();
28 }
29 }
30 }
```

7. (SalidaPantalla2) Indica qué se mostrará por pantalla cuando se ejecute esta clase y por qué:

```

1 public class Dos {
2 private static int metodo() {
3 int valor=0;
4 try {
5 valor = valor+1;
6 valor = valor + Integer.parseInt("W");
7 valor = valor + 1;
8 System.out.println("Valor al final del try: " + valor);
9 } catch(NumberFormatException e) {
10 valor = valor + Integer.parseInt("42");
11 System.out.println("Valor al final del catch: " + valor) ;
12 }
13 finally {
14 valor = valor + 1;
15 System.out.println("Valor al final de finally: " + valor) ;
16 }
17 valor = valor + 1;
18 System.out.println ("Valor antes del return: " + valor) ;
19 return valor ;
20 }
21
22 public static void main (String[] args) {
23 try {
24 System .out.println(metodo());
25 } catch (Exception e) {
26 System.err.println("Excepcion en metodo() ");
27 e.printStackTrace();
28 }
29 }
```

8. (SalidaPantalla3) Indica qué se mostrará por pantalla cuando se ejecute esta clase y por qué:

```

1 public class Tres {
2 private static int metodo() {
3 int valor = 0;
4 try {
5 valor = valor +1;
6 valor = valor + Integer.parseInt("W");
7 valor = valor + 1;
8 System.out.println("Valor al final del try : " + valor);
9 } catch (NumberFormatException e) {
10 valor = valor + Integer.parseInt("W");
11 System.out.println("Valor al final del catch : " + valor);
12 } finally {
13 valor = valor + 1;
14 System.out.println("Valor al final de finally: " + valor);
15 }
16 valor = valor + 1;
17 System.out.println ("Valor antes del return: " + valor);
18 return valor ;
19 }
20
21 public static void main (String[] args)
22 {
23 try {
24 System.out.println(metodo ());
25 } catch (Exception e) {
26 System.err.println("Excepcion en metodo()");
27 e.printStackTrace();
28 }
29 }
30 }

```

9. (SalidaPantalla4) Indica qué se mostrará por pantalla cuando se ejecute esta clase y por qué:

```

1 import java.io.*;
2
3 public class Cuatro
4 {
5 private static int metodo() {
6 int valor = 0;
7 try {
8 valor = valor+1;
9 valor = valor + Integer.parseInt("W");
10 valor = valor + 1;
11 System.out.println("Valor al final del try : " + valor) ;
12 throw new IOException();
13 } catch (IOException e) {
14 valor = valor + Integer.parseInt("42");
15 System.out.println("Valor al final del catch : " + valor);
16 } finally {
17 valor = valor + 1;
18 System.out.println("Valor al final de finally: " + valor);
19 }
20 valor = valor + 1;
21 System.out.println ("Valor antes del return: " + valor) ;
22 return valor ;
23 }
24
25 public static void main(String[] args) {
26 try {
27 System.out.println(metodo ());
28 } catch (Exception e) {
29 System.err.println("Excepcion en metodo()");
30 e.printStackTrace();
31 }
32 }
33 }

```

10. (SalidaPantalla5) Indica qué se mostrará por pantalla cuando se ejecute esta clase:

- a. Si se ejecuta con `java Cinco casa`
- b. Si se ejecuta con `java Cinco 0`
- c. Si se ejecuta con `java Cinco 7`

```

1 public class Cinco {
2 public static void main(String args[]) {
3 try {
4 int a = Integer.parseInt(args[0]);
5 System.out.println("a = " + a);
6 int b=42/a;
7 String c = "holo";
8 char d = c.charAt(50);
9 } catch (ArithmetricException e) {
10 System.out.println("div por 0: " + e);
11 } catch (IndexOutOfBoundsException e) {
12 System.out.println("índice del String fuera de límites: " + e);
13 } finally {
14 System.out.println("Ejecución de finally");
15 }
16 }
17 }
```

11. (SalidaPantalla6) Indica cuál será la salida del siguiente programa y por qué

```

1 public class Seis {
2 public static void procA() {
3 try {
4 System.out.println("dentro del procA"); 2
5 throw new RuntimeException("demo"); 3
6 } finally {
7 System.out.println("Finally del procA"); 4
8 }
9 }
10
11 public static void procB() {
12 try {
13 System.out.println("dentro del procB"); 6
14 return; 7
15 } finally {
16 System.out.println("finally del procB"); 8
17 }
18 }
19
20 public static void main(String args[]) {
21 try {
22 procA(); 1
23 } catch(Exception e) {
24 procB(); 5
25 }
26 }
27 }
```

12. (SalidaPantalla7) Indica cuál será la salida del siguiente programa y por qué

```

1 public class Siete {
2 public static void metodo() {
3 try {
4 throw new NullPointerException("demo"); 2
5 } catch (NullPointerException e) {
6 System.out.println("capturada en método"); 3
7 throw e; 4
8 }
9 }
10
11 public static void main (String args[]) {
12 try {
13 metodo(); 1
14 } catch(NullPointerException e) {
15 System.out.println("capturada en main " + e); 5
16 }
17 }
18 }
```

13. (DivisionPorCero) Crea un programa que intente dividir dos números enteros ingresados por el usuario y maneja la excepción de división por cero. [Aquí](#) tienes la explicación de porqué la división entre 0 no provoca excepciones para `double` y `float`.

```

1 Introduce el dividendo (entero): 85
2 Introduce el divisor (entero): 4
3 El resultado de 85/4 es: 21
4
5 Introduce el dividendo (entero): 5
6 Introduce el divisor (entero): 0
7 Error: División por cero no permitida.
```

14. (CalculadoraExcepcion) Crea una clase `calculadoraExcepcion` con un método `dividir` que acepte dos números con decimales como argumentos, devuelva su resultado o lance una excepción personalizada (`DivisionPorCeroException`) si el divisor es cero. Captura la excepción en el método `main` y muestra un mensaje de error.

```

1 Introduce el dividendo: 15,5
2 Introduce el divisor: 4
3 El resultado de 15,5/4,0 es: 3,875
4
5 Introduce el dividendo: 3,5
6 Introduce el divisor: 0
7 Error: División por cero no permitida.

```

15. (EntradaNoNumerica) Escribe un programa que lea un número entero desde el teclado. Si el usuario ingresa algo que no es un número entero, maneja la excepción y muestra un mensaje de error.
16. (RangoNumerico) Escribe un programa que solicite al usuario ingresar un número entre 1 y 100. Si el número está fuera de ese rango, lanza una excepción personalizada (`NúmeroFueraDeRangoException`) y muestra un mensaje de error.

```

1 Introduce un número entre 1 y 100: 5
2 Número válido: 5
3
4 Introduce un número entre 1 y 100: 500
5 Error: Número fuera de rango: 500

```

17. (NumeroNegativo) Crea un método que reciba dos números como argumentos y lance una excepción personalizada si uno de los números es negativo. Captura esa excepción en el método principal y muestra un mensaje de error.

```

1 Introduce el primer número: 5
2 Introduce el segundo número: 4
3 Ambos números son positivos.
4
5 Introduce el primer número: -4
6 Introduce el segundo número: 5
7 Error: Uno de los números es negativo.

```

18. (LongitudCadena) Diseña un programa que lea una cadena de caracteres desde el teclado y, si la longitud de la cadena es mayor de 10 caracteres, lance una excepción personalizada. Captura esa excepción y muestra un mensaje de error.
19. (TemperaturaInvalida) Implementa una clase `ConversorTemperatura` que tenga un método para convertir grados Celsius a Fahrenheit. Si el valor en grados Celsius es inferior a -273.15, lanza una excepción personalizada. Captura la excepción y muestra un mensaje de error en el método principal.
20. (EdadInvalida) Diseña una clase `ValidadorEdad` que tenga un método para validar si una persona tiene una edad válida (por ejemplo, entre 0 y 120 años). Si la edad no es válida, lanza una excepción personalizada y muestra un mensaje de error en el método principal.

### 10.2.8. Aserciones

1. (Aserciones1) A partir del siguiente fragmento de código, añade una linea debajo del comentario de la linea 4 que haga lo que se solicita:

```

1 class Main {
2 public static void main(String args[]) {
3 String[] finde = {"viernes", "sabado", "domingo"};
4 //Añade una aserción que compruebe que solo hay dos días en el fin de semana.
5
6 System.out.println("Solo hay " + weekends.length + " días en el fin de semana");
7 }
8 }

```

2. (Aserciones2) Escribe un método llamado `validarEdad(int edad)` que acepte como parámetro la edad de una persona. Usa una aserción para verificar que la edad sea un valor positivo y menor que 150. Si la edad es negativa o extremadamente alta, la aserción debería fallar. Dispones del siguiente `main`:

```

1 public static void main(String[] args) {
2 // Caso que debe pasar la aserción
3 validarEdad(25);
4 // Caso que debe fallar la aserción (edad negativa)
5 validarEdad(-5);
6 }

```

3. (Aserciones3) Crea un método llamado `esPar(int numero)` que devuelva `true` si el número es par y `false` en caso contrario. Luego, escribe una aserción para verificar que el resultado es `true` cuando el número proporcionado es efectivamente par. Dispones del siguiente `main`:

```

1 public static void main(String[] args) {
2 // Caso que debe devolver true
3 assert esPar(4) : "El número 4 debería ser par";
4 // Caso que debe devolver false
5 assert !esPar(3) : "El número 3 no debería ser par";
6 // Hace saltar la excepción si las aserciones están activadas y alguna falla
7 assert esPar(5) : "Hará saltar la excepción";
8 }

```

4. (Aserciones4) Implementa un método llamado `dentroDeRango(int numero, int min, int max)` que devuelva `true` si el número está en el rango `[min, max]` y `false` en caso contrario. Usa aserciones para probar que el método devuelve `true` para un número dentro del rango y `false` para uno fuera.

```

1 // Ejemplo de uso:
2 assert dentroDeRango(5, 1, 10) : "El número 5 debería estar en el rango [1, 10]";
3 assert !dentroDeRango(15, 1, 10) : "El número 15 no debería estar en el rango [1, 10]";

```

## 10.3. Actividades

1. (TransformarBucle) Transforma el siguiente bucle for en un bucle while:

```

1 for (i=5; i<15; i++) {
2 System.out.println(i);
3 }

```

2. (NumerosPares) Escribe un método (`nPares`) que recibe un número `n` y que devuelve un `String` con los `n` primeros números pares. Escribe un método `main` que pida un número al usuario y use el método `nPares` para mostrarlos en pantalla.

```

1 Introduce un número entero positivo: 15
2 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30

```

3. (Rango) Escribe un método (`rango`) que recibe dos enteros y que devuelve un `String` que comience con el primero y termine en el segundo. Se asume que el primero será menor que el segundo. Escribe un método `main` que pida los dos números al usuario y muestre el rango de números usando el método `rango`:

```

1 Introduce el primer número del rango: 200
2 Introduce el segundo número del rango: 250
3 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 2
38 239 240 241 242 243 244 245 246 247 248 249 250

```

4. (TablasMultiplicar) Escribe un método (`tabla`) que recibe un número entero positivo y devuelve un `String` con la tabla de multiplicar de dicho número con el siguiente formato:

```

1 Tabla del 2
2 2 x 1 = 2
3 2 x 2 = 4
4 ...
5 2 x 10 = 20

```

Escribe un método `main`, que usando el método `tabla` imprima por pantalla las tablas del 1 al 10.

5. (SinMultiplos5) Programa que muestre los números del 1 al 100 sin mostrar los múltiplos de 5.

```

1 ...
2 28
3 29
4 31
5 32
6 33
7 34
8 36
9 37
10 38
11 39
12 41
13 42
14 43
15 ...

```

**A**cción

Para pasar satisfactoriamente los tests, en lugar de usar `System.out.println`, debes usar `System.out.print` con un `\n` al final de la cadena a imprimir.

6. (CuadradoHastaNegativo) Leer un número y mostrar su cuadrado, repetir el proceso hasta que se introduzca un número negativo.
7. (PositivoNegativo) Leer un número e indicar si es positivo o negativo. El proceso se repetirá hasta que se introduzca un 0.
8. (ParImpar) Leer números hasta que se introduzca un 0. Para cada uno indicar si es par o impar.
9. (ContarNumeros) Pedir números hasta que se teclee uno negativo, y mostrar cuántos números se han introducido.
10. (AdivinarNúmero) Realizar un juego para adivinar un número  $x$ . Para ello pedir un número  $N$ , y luego ir pidiendo números indicando "mayor" o "menor" según sea mayor o menor con respecto a  $x$ . El proceso termina cuando el usuario acierta.

```

1 Voy a pensar un número menor que 1000 para que lo adivines...
2
3 Introduce que número crees que he pensado: 500
4 El número que he pensado es menor que 500
5 Introduce que número crees que he pensado: 250
6 El número que he pensado es menor que 250
7 Introduce que número crees que he pensado: 125
8 El número que he pensado es mayor que 125
9 Introduce que número crees que he pensado: 187
10 El número que he pensado es mayor que 187
11 Introduce que número crees que he pensado: 210
12 El número que he pensado es mayor que 210
13 Introduce que número crees que he pensado: 230
14 El número que he pensado es mayor que 230
15 Introduce que número crees que he pensado: 240
16 El número que he pensado es mayor que 240
17 Introduce que número crees que he pensado: 245
18 Enhorabuena has acertado, el número que había pensado es el 245

```

11. (SumaNumeros) Pedir números hasta que se teclee un 0, mostrar la suma de todos los números introducidos.
12. (MediaNumeros) Pedir números hasta que se introduzca uno negativo, y calcular la media.
13. (NumerosHastaN) Pedir un número  $N$ , y mostrar todos los números del 1 al  $N$ .
14. (De100a0) Escribir todos los números del 100 al 0 de 7 en 7.
15. (Suma15Numeros) Pedir 15 números y escribir la suma total.
16. (ProductoImpares) Diseñar un programa que muestre el producto de los 10 primeros números impares.
17. (Factorial) Pedir un número y calcular su factorial (el factorial se representa con el simbolo  $!$ ).

Aquí tienes el factorial de los 5 primeros números enteros:

```

1 1! = 1
2 2! = 2 * 1 = 2
3 3! = 3 * 2 * 1 = 6
4 4! = 4 * 3 * 2 * 1 = 24
5 5! = 5 * 4 * 3 * 2 * 1 = 120

```

Ejemplo de ejecución del programa:

```

1 Dime el número para calcular su factorial: 6
2 6! = 6 * 5 * 4 * 3 * 2 * 1 = 720

```

18. (MediaPosNeg) Pedir 10 números. Mostrar la media de los números positivos, la media de los números negativos y la cantidad de ceros.
19. (Sueldos1000) Pedir 10 sueldos. Mostrar su suma y cuantos hay mayores de 1000€.
20. (AlumnosEdadAltura) Dadas las edades y alturas de 5 alumnos, mostrar la edad y la estatura media, la cantidad de alumnos mayores de 18 años, y la cantidad de alumnos que miden más de 1.75.
21. (TablaMultiplicar) Pide un número (que debe estar entre 0 y 10) y mostrar la tabla de multiplicar de dicho número.
22. (AprobadosSuspensos) Dadas 6 notas, escribir la cantidad de alumnos aprobados y suspensos.
23. (SueldoMaximo) Pedir un número  $N$ , introducir  $N$  sueldos, y mostrar el sueldo máximo.
24. (HayNegativo) Pedir 10 números, y mostrar al final si se ha introducido alguno negativo.
25. (HaySuspensos) Pedir 5 calificaciones de alumnos y decir al final si hay algún suspensos.
26. (Multiplo3) Pedir 5 números e indicar si alguno es múltiplo de 3.

27. (SaludoHorario) Realiza un programa que pida una hora por teclado y que muestre luego buenos días, buenas tardes o buenas noches según la hora. Se utilizarán los tramos de 6 a 12, de 13 a 20 y de 21 a 5. respectivamente. Sólo se tienen en cuenta las horas, los minutos no se deben introducir por teclado.
28. (DiaSemana) Escribe un programa en que dado un número del 1 a 7 escriba el correspondiente nombre del día de la semana.
29. (SalarioHorasExtras) Escribe un programa que calcule el salario semanal de un trabajador teniendo en cuenta que las horas ordinarias (40 primeras horas de trabajo) se pagan a 12 euros la hora. A partir de la hora 41, se pagan a 16 euros la hora.
30. (MediaTresNotas) Realiza un programa que calcule la media de tres notas.
31. (BoletinNotas) Amplía el programa anterior para que diga la nota del boletín (insuficiente, suficiente, bien, notable o sobresaliente).
32. (Horoscopo) Escribe un programa que nos diga el horóscopo a partir del día y el mes de nacimiento.
33. (Cuestionario) Realiza un minicuestionario con 4 preguntas tipo test sobre las asignaturas que se imparten en el curso. Cada pregunta acertada sumará un punto. El programa mostrará al final la calificación obtenida.
34. (NotaProgramacion) Calcula la nota de un trimestre de la asignatura Programación. El programa pedirá las dos notas que ha sacado el alumno en los dos primeros controles. Si la media de los dos controles da un número mayor o igual a 5, el alumno está aprobado y se mostrará la media. En caso de que la media sea un número menor que 5, el alumno habrá tenido que hacer el examen de recuperación que se califica como apto o no apto, por tanto se debe preguntar al usuario ¿Cuál ha sido el resultado de la recuperación? (apto/no apto). Si el resultado de la recuperación es apto, la nota será un 5; en caso contrario, la nota será 1.

Ejemplo 1:

```
1 Nota del primer control: 7 Nota del segundo control: 10
2 Tu nota de Programación es 8.5
```

Ejemplo 2:

```
1 Nota del primer control: 6 Nota del segundo control: 3
2 ¿Cuál ha sido el resultado de la recuperación? (apto/no apto): apto
3 Tu nota de Programación es 5
```

Ejemplo 3:

```
1 Nota del primer control: 6 Nota del segundo control: 3
2 ¿Cuál ha sido el resultado de la recuperación? (apto/no apto): no apto
3 Tu nota de Programación es 1
```

35. (Multiplos5For) Muestra los números múltiplos de 5 entre el 0 y el 100 utilizando un bucle `for`.
36. (Multiplos5While) Muestra los números múltiplos de 5 entre el 0 y el 100 utilizando un bucle `while`.
37. (Multiplos5DoWhile) Muestra los números múltiplos de 5 entre el 0 y el 100 utilizando un bucle `do while`.
38. (ContarAtrasFor) Muestra los números del 320 al 160, contando de 20 en 20 hacia atrás utilizando un bucle `for`.
39. (ContarAtrasWhile) Muestra los números del 320 al 160, contando de 20 en 20 hacia atrás utilizando un bucle `while`.
40. (ContarAtrasDoWhile) Muestra los números del 320 al 160, contando de 20 en 20 utilizando un bucle `do-while`.
41. (CajaFuerte) Realiza el control de acceso a una caja fuerte. La combinación será un número de 4 cifras. El programa nos pedirá la combinación para abrirla. Si no acertamos, se nos mostrará el mensaje "**Esa no es la combinación correcta**" y si acertamos se nos dirá "**La caja se ha abierto correctamente**". Tendremos cuatro oportunidades para abrir la caja fuerte, si lo sobrepasamos nos dirá "**Has sobrepasado el número de intentos permitido**".

### Acción

Para pasar satisfactoriamente los tests, la combinación de apertura debe estar configurada en "1234"

42. (CuadradoCubo) Escribe un programa que muestre en tres columnas, el cuadrado y el cubo de los 5 primeros números enteros a partir de uno que se introduce por teclado.
43. (Potencia) Escribe un método (`potencia`) que reciba una base y un exponente (enteros positivos) y que calcule la potencia (sin usar `Math`). Escribe un método `main` para pedir los datos al usuario y usando el método `potencia` mostrar el resultado.

```
1 Vamos a calcular una potencia.
2 Introduce la BASE: 6
3 Introduce el EXPONENTE: 5
4 El resultado de elevar la BASE: 6 al EXPONENTE: 5 resulta en: 7776
```

44. (Suma100Siguentes) Realiza un programa que sume los 100 números siguientes a un número entero y positivo introducido por teclado. Se debe comprobar que el dato introducido es correcto (que es un número positivo).
45. (NumerosEntre7) Escribe un programa que imprima por pantalla los números enteros comprendidos entre dos números introducidos por teclado y validados como distintos, el programa debe empezar por el menor de los enteros introducidos e ir incrementando de 7 en 7.

```

1 Introduce dos números DIFERENTES!
2 Introduce el primer número: 8
3 Introduce el segundo número: 8
4 Los números no son DIFERENTES!
5
6 Introduce dos números DIFERENTES!
7 Introduce el primer número: 6
8 Introduce el segundo número: 50
9 6
10 13
11 20
12 27
13 34
14 41
15 48

```

46. (EstadisticasNumeros) Realiza un programa que vaya pidiendo números hasta que se introduzca un numero negativo y nos diga cuantos números se han introducido, la media de los impares y el mayor de los pares. El número negativo sólo se utiliza para indicar el final de la introducción de datos pero no se incluye en el cómputo.
47. (SumaHasta10000) Escribe un programa que permita ir introduciendo una serie indeterminada de números mientras su suma no supere el valor 10000. Cuando esto último ocurra, se debe mostrar el total acumulado, el contador de los números introducidos y la media.
48. (Multiplos3) Escribe un programa que muestre, cuente y sume los múltiplos de 3 que hay entre 1 y un número leído por teclado.
49. (PrecioFinal) Escribe un programa que calcule el precio final de un producto según su base imponible (precio antes de impuestos), el tipo de IVA aplicado (general, reducido o superreducido) y el código promocional. Los tipos de IVA general, reducido y superreducido son del 21%, 10% y 4% respectivamente. Los códigos promocionales pueden ser `nopro`, `mitad`, `menos5` o `5porc` que significan respectivamente que no se aplica promoción, el precio se reduce a la mitad, se descuentan 5 euros o se descuenta el 5%.

Ejemplo:

```

1 Introduzca la base imponible: 25
2 Introduzca el tipo de IVA (general, reducido o superreducido): reducido
3 Introduzca el código promocional (nopro, mitad, menos5 o 5porc): mitad
4 Base imponible 25.00
5 Cód. promo. (mitad): -12.50
6 IVA (10%) 1.25
7 Precio con IVA 13.75
8 TOTAL 13.75

```

50. (AnioBisiesto) Pedir un año e indicar si es bisiesto, teniendo en cuenta que son bisiestos todos los años divisibles por 4, excluyendo los que sean divisibles por 100, pero no los que sean divisibles por 400.

En pseudocódigo se calcularía así:

```

1 SI ((año divisible por 4) Y ((año no divisible por 100) O (año divisible por 400)))ENTONCES
2 es bisiesto
3 SINO
4 no es bisiesto
5 FIN_SI

```

51. (NumeroALetras) Pedir un número de 20 a 99 y mostrarlo escrito. Por ejemplo, para 56 mostrar: cincuenta y seis.
52. (VehiculoIVA) Introducir datos de un vehículo (marca, modelo y precio). Devolver el precio con IVA del vehículo. Controlar con Excepciones que el precio del vehículo introducido son números y que el cálculo de Precio Final con IVA no devuelva error.
53. (NotaMediaAlumnos) Introducir códigos de alumnos, nombre y nota hasta que se introduzca un código de alumno negativo. Devolver la nota media de los alumnos la clase. Controlar con Excepciones que las notas introducidas son números y que si no se introducen alumnos el cálculo de la media no devuelva error.
54. (ImporteFinal) Crear una función o método llamado `impFinal`, que calcule el importe final de una compra. Los parámetros que se le pasarán a la función son el `precio` del producto, las `cantidad` de unidades compradas, el `porcentaje` de `iva` y el `porcentaje` de `descuento`. El método principal debe pedir por teclado el precio del producto, las unidades adquiridas, el porcentaje de IVA y el porcentaje de descuento y devolver el `Importe final` de la Factura.
55. (CapacidadDisco) Crear una función que calcule la capacidad de un disco. La capacidad se calcula multiplicando los Cabezales o pistas del disco por los Cilindros por los Sectores por Tamaño de Sector. El método principal debe pedir por teclado los Cabezales o Pistas del disco, los Cilindros, Sectores y Tamaño de Sector y devolver la Capacidad del disco en Gigabytes.

### Ejemplo

Por ejemplo: Calcular la capacidad de un disco teniendo en cuenta que dispone de 10 Cabezales o Pistas, 65535 Cilindros, 1024 Sectores/pista y un Tamaño de 512 bytes/sector:

$$\text{Capacidad del disco} = 10 * 65535 * 1024 * 512 = 343597383680 \text{ bytes}$$

$$343597383680 \text{ bytes} / 1024 / 1024 / 1024 = 320 \text{ Gbytes}$$

56. (MayorDeTres) Función que devuelva el mayor de tres números. El método principal debe pedir por teclado los tres números introducidos por el teclado. La función debe recibir como parámetros los tres números y devolver el mayor.

4 de diciembre de 2025

## 11. 4.3 Talleres

---

### 11.1. Taller UD03\_01: GitHub Classroom

#### **11.1.1. Unirnos a GitHub Classroom**

Aceptamos el *Assignement* (la tarea/ejercicio) a partir del link del profesor, en este caso: <https://classroom.github.com/a/9bM3kTmM>

#### **11.1.2. Tarea**

Debes enviar tus soluciones a GitHub Classroom y superar al menos la mitad de los tests, cuantos más tests superados, mejor nota tendrás en la tarea.

 21 de noviembre de 2025

## 11.2. Taller UD03\_02: Acepta el reto

---

### **11.2.1. Inscríbete en Acepta el reto**

Inscríbete en la web [www.aceptaelreto.com](http://www.aceptaelreto.com) seleccionando nuestro instituto.

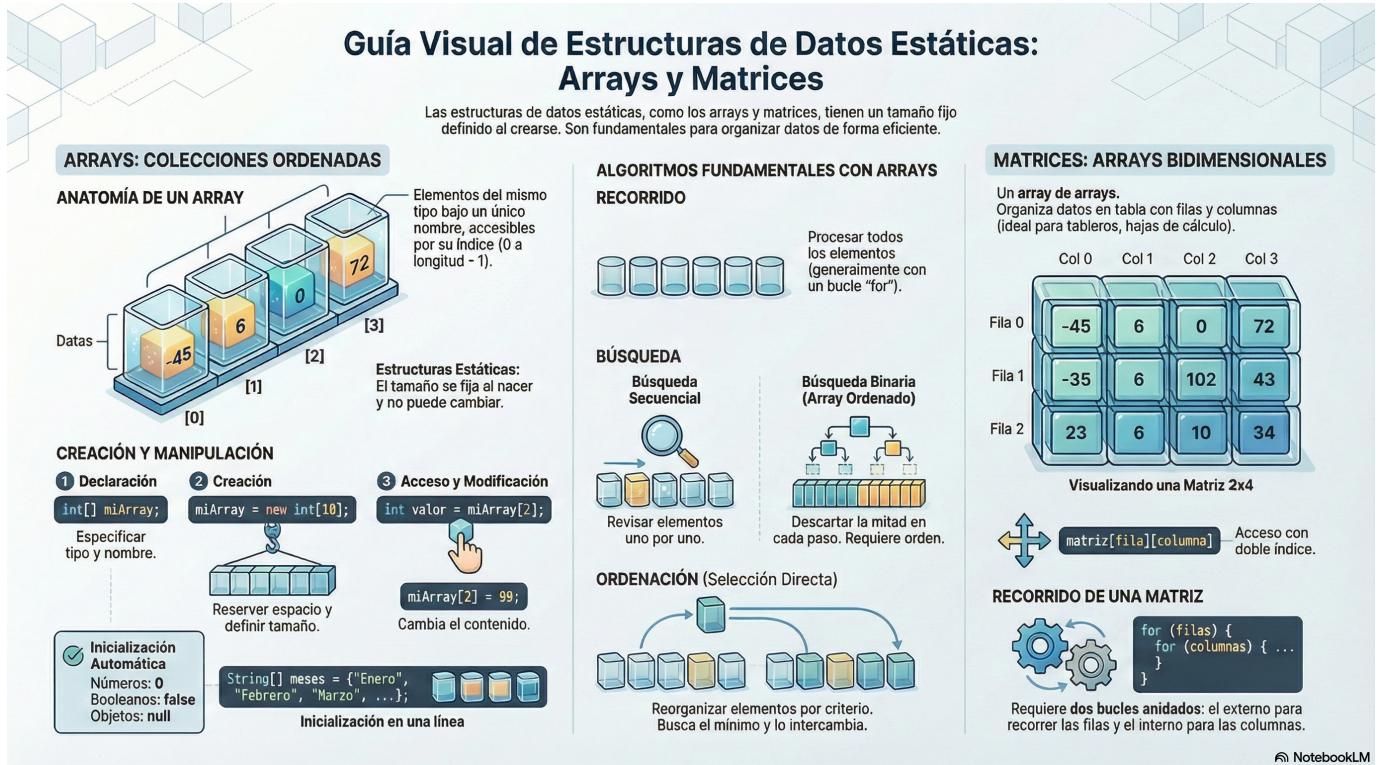
### **11.2.2. Tarea**

Haz una captura de pantalla de tu perfil una vez hecho login con tu usuario y envia la captura en un pdf al profesor.

 21 de noviembre de 2025

## 5. UD04

### 12. 5.1 Estructuras de datos: Arrays y matrices. Recursividad.



#### 12.1. Introducción

A menudo, para resolver problemas de programación, no basta con disponer de sentencias condicionales o iterativas como las que hemos visto (`if`, `switch`, `while`, `for`, ...).

También es necesario disponer de herramientas para organizar la información de forma adecuada: **las estructuras de datos**.

Los arrays son una estructura de datos fundamental, que está disponible en la mayoría de lenguajes de programación y que nos permitirá resolver problemas que, sin ellos, resultarían difíciles o tediosos de solucionar.

Imaginemos, por ejemplo, que queremos leer los datos de pluviosidad de cada uno de los 31 días de un mes. Posteriormente se desea mostrar la pluviosidad media del mes y en cuántos días las lluvias superaron la media.

Con las herramientas de que disponemos hasta ahora, nos veríamos obligados a declarar **31 variables double**, una para cada día, y a elaborar un largo programa que leyera los datos y contara cuáles superan la media. Con el uso de arrays, problemas como este tienen una solución fácil y corta.

#### 12.2. Arrays

Un array es una colección de elementos del mismo tipo, que tienen un nombre o identificador común.

Se puede acceder a cada componente del array de forma individual para consultar o modificar su valor. El acceso a los componentes se realiza mediante un subíndice, que viene dado por la posición que ocupa el elemento dentro del array.

En la siguiente figura se muestra un array `c` de enteros:

|   |      |      |      |      |      |      |      |      |      |      |
|---|------|------|------|------|------|------|------|------|------|------|
| c | -1   | 8    | 23   | 5    | 12   | -5   | 255  | -28  | 30   | 42   |
|   | c[0] | c[1] | c[2] | c[3] | c[4] | c[5] | c[6] | c[7] | c[8] | c[9] |

**Importante**

El primer subíndice de un array es el cero. El último subíndice es la longitud del array menos uno.

El número de componentes de un array se establece inicialmente al crearlo y no es posible cambiarlo de tamaño. Es por esto que reciben el nombre de estructuras de datos estáticas.

**12.2.1. Declaración y creación**

Para poder utilizar un array hay que declararlo:

```
1 tipo nombreVariable[];
```

o

```
1 tipo[] nombreVariable;
```

En la declaración se establece el nombre de la variable y el tipo de los componentes. Por ejemplo:

```
1 double lluvia1[]; // lluvia1 es un array de double
2 double[] lluvia2; // lluvia2 es un array de double <== Esta es la declaración recomendada, el [] siempre acompañando al tipo.
```

En la declaración anterior no se ha establecido el número de componentes. El número de componentes se indica en la creación, que se hace utilizando el operador `new`:

```
1 lluvia1 = new double[31];
```

Con esta instrucción se establece que el número de elementos del array `lluvia` son 31, reservando con ello el compilador espacio consecutivo para 31 componentes individuales de tipo `double`.

Las dos instrucciones anteriores se pueden unir en una sola:

```
1 // tipo[] nombreVariable = new tipo[numElementos];
2 double[] lluvia2 = new double[31];
```

El valor mediante el cual se define el número de elementos del array tiene que ser una expresión entera, pero no tiene por qué ser un literal como en el ejemplo anterior. El tamaño de un array se puede establecer durante la ejecución, como en el siguiente ejemplo:

```
1 // usamos un array para almacenar las edades de un grupo de personas
2 // la variable numPersonas contiene el número de personas del grupo
3 // y se asigna en tiempo de ejecución
4 Scanner teclado = new Scanner(System.in);
5 System.out.print("Introduce cuantos elementos debe tener el array edad[]:");
6 int numPersonas = teclado.nextInt();
7 int[] edad = new int[numPersonas];
```

**12.2.2. Acceso a los componentes**

Como ya hemos dicho, el acceso a los componentes del array se realiza mediante subíndices. La sintaxis para referirse a un componente del array es la siguiente:

```
1 nombreVariable[subíndice]
```

Tras declarar el array `lluvia`, se dispone de 31 componentes de tipo `double` numeradas desde la 0 a la 30 y accesibles mediante la notación: `lluvia[0]` (componente primera), `lluvia[1]` (componente segunda) y así sucesivamente hasta la última componente: `lluvia[30]`.

Con cada una de las componentes del array de `double` `lluvia` es posible efectuar todas las operaciones que podrían realizarse con variables individuales de tipo `double`, por ejemplo, dadas las declaraciones anteriores, las siguientes instrucciones serían válidas:

```

1 int[] edad = new int[3];
2 System.out.print("Introduce el dato para el componente 0: ");
3 edad[0] = teclado.nextInt(); //25
4 System.out.println("El componente [0] vale " + edad[0]);
5 edad[1] = edad[0] + 1;
6 edad[2] = edad[0] + edad[1];
7 edad[2]++;
8 System.out.println("El componente [1] vale " + edad[1]); //26
9 System.out.println("El componente [2] vale " + edad[2]); //52

```

Además, hay que tener en cuenta que el subíndice ha de ser una expresión entera, por lo que también son válidas expresiones como las siguientes:

```

1 int i;
2 ...
3 edad[i] = edad[i + 1];
4 edad[i + 2] = edad[i];

```

### 12.2.3. Inicialización

Cuando creamos un array, Java inicializa automáticamente sus componentes:

- Con 0 cuando los componentes son de tipo numérico.
- Con false cuando los componentes son `boolean`.
- Con el carácter de ASCII 0, cuando los componentes son `char`.
- Con `null` cuando son objetos (`Strings`, etc)

Aun así, es probable que estos no sean los valores con los que queremos inicializar el array. Tenemos entonces dos posibilidades:

- Acceder individualmente a los componentes del array para darles valor:

```

1 int edad2[] = new int[10];
2 edad2[0] = 25;
3 edad2[1] = 10;
4 ...
5 edad2[9] = 12;

```

- O inicializar el array en la declaración de la siguiente forma:

```
1 int edad3[] = {25, 10, 23, 34, 65, 23, 1, 67, 54, 12};
```

#### Más información

Es decir, enumerando los valores con los que se quiere inicializar cada componente, encerrados entre llaves. De hacerlo así, no hay que crear el array con el operador `new`. Java crea el array con tantos componentes como valores hemos puesto entre llaves. Además no es necesario indicar el número de elementos del mismo.

### 12.2.4. Un ejemplo práctico

Ya hemos resuelto en temas anteriores el problema de devolver el nombre de un mes dado su número.

Vamos a resolverlo ahora ayudándonos de arrays:

```

1 public static String nombreMes(int mes){
2 String[] nombre = {"enero", "febrero", "marzo", "abril",
3 "mayo", "junio", "julio", "agosto",
4 "septiembre", "octubre", "noviembre", "diciembre"};
5 return nombre[mes-1];
6 }

```

El método define un array de `String` que se inicializa con los nombres de los doce meses. La primera componente del array (`nombre[0]`) se deja vacía, de forma que enero quede almacenado en `nombre[1]`. Devolver el nombre del mes indicado se reduce a devolver el componente del array cuyo número indica el parámetro mes: `nombre[mes]`

#### 12.2.5. Arrays como parámetros. Paso de parámetros por referencia.

Hasta el momento sólo se ha considerado el paso de parámetros por valor; de manera que cualquier cambio que el método realice sobre los parámetros formales no modifica el valor que tiene el parámetro real con el que se llama al método. En java, todos los parámetros de tipo simple (`byte`, `short`, `int`, ...) se pasan por valor.

##### Importante

Por el contrario, los arrays no son variables de tipo primitivo, y como cualquier otro objeto, se pasa siempre por referencia.

En el paso de parámetros por referencia lo que se pasa en realidad al método es la dirección de la variable u objeto. Es por esto que el papel del parámetro formal es el de ser una referencia al parámetro real; la llamada al método no provoca la creación de una nueva variable. De esta forma, las modificaciones que el método pueda realizar sobre estos parámetros se realizan efectivamente sobre los parámetros reales. En este caso, ambos parámetros (formal y real) se pueden considerar como la misma variable con dos nombres, uno en el método llamante y otro en el llamado o invocado, pero hacen referencia a la misma posición de memoria.

En el siguiente ejemplo, la variable `a`, de tipo primitivo, no cambia de valor tras la llamada al método. Sin embargo la variable `v`, array de enteros, si se ve afectada por los cambios que se han realizado sobre ella en el método:

```

1 public static void main(String[] args){
2 int a = 1;
3 int[] v = {1,1,1};
4 metodo(v,a); //Pasar un array como parámetro
5 System.out.println(a); // Muestra 1
6 System.out.println(v[0]); // Muestra 2
7 }
8
9 public static void metodo(int[] x, int y){ //recibir un array como parámetro
10 x[0]++;
11 y++;
12 }
```

##### Importante

Como podemos observar, para pasar un array a un método, simplemente usamos el nombre de la variable en la llamada. En la cabecera del método, sin embargo, tenemos que utilizar los corchetes `[]` para indicar que el parámetro es un array.

#### 12.2.6. El atributo `length`

Todas las variables de tipo array tienen un atributo `length` que permite consultar el número de componentes del array. Su uso se realiza posponiendo `.length` al nombre de la variable:

```

1 double[] estatura = new double[25];
2 ...
3 System.out.println(estatura.length); // Mostrará por pantalla: 25
```

#### 12.2.7. `String[] args` en el `main`

El método `main` puede recibir argumentos desde la línea de comandos. Para ello, el método `main` recibe un parámetro (`String args[]`). Vemos que se trata de un array de `Strings`. El uso del atributo `length` nos permite comprobar si se ha llamado al programa de forma correcta o no. Veamos un ejemplo para saber si es Navidad. Se habrá llamado correctamente si el array `args` contiene dos componentes (día, mes):

```

1 public class EsNavidad {
2 public static void main(String[] args) {
3 if (args.length != 2) {
4 System.out.println("ERROR:");
5 System.out.println("Llame al programa de la siguiente forma:");
6 System.out.println("java EsNavidad dia mes");
7 } else {
8 // args[0] es el dia
9 // args[1] es el mes
10 if ((Integer.valueOf(args[0]) == 25) && (Integer.valueOf(args[1]) == 12)) {
11 System.out.println("ES NAVIDAD!");
12 } else {
13 System.out.println("No es navidad.");
14 }
15 }
16 }
17 }
```

## 12.3. Problemas de recorrido, búsqueda y ordenación

Muchos de los problemas que se plantean cuando se utilizan arrays pueden clasificarse en tres grandes grupos de problemas genéricos: los que conllevan el recorrido de un array, los que suponen la búsqueda de un elemento que cumpla cierta característica dentro del array, y los que implican la ordenación de los elementos del array.

La importancia de este tipo de problemas proviene de que surgen, no sólo en el ámbito de los arrays, sino también en muchas otras organizaciones de datos de uso frecuente (como las listas, los ficheros, etc.). Las estrategias básicas de resolución que se verán a continuación son también extrapolables a esos otros ámbitos.

### 12.3.1. Problemas de recorrido

Se clasifican como problemas de recorrido aquellos que para su resolución exigen algún tratamiento de todos los elementos del array. El orden para el tratamiento de estos elementos puede organizarse de muchas maneras: ascendente, descendente, ascendente y descendente de forma simultánea, etc.

En el siguiente ejemplo se muestra un método en java para devolver, a partir de un array que contiene la pluviosidad de cada uno de los días de un mes, la pluviosidad media de dicho mes. Para ello se recorren ascendenteamente los componentes del array para ir sumándolos:

```

1 public static double pluviosidadMediaAscendente(double[] lluvia){
2 double suma = 0;
3 //Recorremos el array ascendente
4 for (int i = 0; i<lluvia.length; i++){
5 suma += lluvia[i];
6 }
7 double media = suma / lluvia.length;
8 return media;
9 }
```

La forma de recorrer el array ascendente es, como vemos, utilizar una variable entera (`i` en nuestro caso) que actúa como subíndice del array. Éste subíndice va tomando los valores `0, 1, ..., lluvia.length-1` en el seno de un bucle, de manera que se accede a todos los componentes del array para sumarlos.

El mismo problema resuelto con un recorrido descendente sería como sigue:

```

1 public static double pluviosidadMediaDescendente(double[] lluvia){
2 double suma = 0;
3 //Recorremos el array descendente
4 for (int i = lluvia.length-1; i>=0; i--){
5 suma += lluvia[i];
6 }
7 double media = suma / lluvia.length;
8 return media;
9 }
```

También realizamos un recorrido para obtener la pluviosidad máxima del mes (la cantidad de lluvia más grande caída en un día), es decir, el elemento más grande del array:

```

1 public static double pluviosidadMaxima(double[] lluvia){
2 // Suponemos que la pluviosidad máxima se produjo el primer día
3 double max = lluvia[0];
4 // Recorremos el array desde la posición 1, comprobando si hay una pluviosidad mayor
5 for (int i = 1; i < lluvia.length; i++) {
6 if(lluvia[i] > max){
7 max = lluvia[i];
8 }
9 }
10 return max;
}

```

#### 12.3.1.1. BUCLE FOR EACH (FOR-LOOP)

En el tema anterior vimos algún tipo de bucles que explicaríamos cuando los pudiésemos utilizar, en este grupo están los bucles for each o for-loops. Aquí tenemos un ejemplo de recorrido de un array con la sintaxis que ya conocemos:

```

1 int[] array = { 1, 2, 3, 4, 5, 6, 7, 8 };
2 for (int i = 0; i < array.length; i++) {
3 System.out.print(array[i] + " ");
4 }

```

el anterior fragmento genera la siguiente salida:

```
1 1 2 3 4 5 6 7 8
```

Este mismo código se puede escribir de la siguiente manera:

```

1 int[] array = { 1, 2, 3, 4, 5, 6, 7, 8 };
2 for (int i : array) { // Mentalmente podemos traducir por:
3 // "para cada entero "i" que encontraremos en el array"
4 System.out.print(i + " ");
5 }

```

la salida seguirá siendo la misma:

```
1 1 2 3 4 5 6 7 8
```



Cuidado!

Con el segundo método no tenemos acceso a la posición o índice del array, este método no serviría para métodos en los que necesitamos conocer la posición o utilizarla de alguna manera. Traducimos el método de `pluviosidadMedia` con un bucle `loop`:

```

1 public static double pluviosidadMediaLoop(double[] lluvia){
2 double suma = 0;
3 // Recorremos el array con el loop
4 for (int i : lluvia){
5 suma += i;
6 }
7 double media = suma / lluvia.length;
8 return media;
9 }

```

#### 12.3.2. Problemas de búsqueda

Se denominan problemas de búsqueda a aquellos que, de alguna manera, implican determinar si existe algún elemento del array que cumpla una propiedad dada. Con respecto a los problemas de recorrido presentan la diferencia de que no es siempre necesario tratar todos los elementos del array: el elemento buscado puede encontrarse inmediatamente, encontrarse tras haber recorrido todo el array, o incluso no encontrarse.

##### 12.3.2.1. BÚSQUEDA ASCENDENTE

Consideremos, por ejemplo, el problema de encontrar cuál fue el primer día del mes en que no llovió nada, es decir, el primer elemento del array con valor cero:

```

1 //Devolveremos el subíndice del primer componente del array cuyo valor es cero.
2 // Si no hay ningún día sin lluvias devolveremos -1
3 public static int primerDiaSinLluvia1(double[] lluvia){
4 int i=0 ;
5 boolean encontrado = false ;
6 while (i<lluvia.length && !encontrado){
7 if (lluvia[i] == 0) encontrado = true ;
8 else i++ ;
9 }
10 if (encontrado) return i ;
11 else return -1 ;
12 }

```

Hemos utilizado el esquema de búsqueda: Definimos una variable `boolean` que indica si hemos encontrado o no lo que buscamos. El bucle se repite mientras no lleguemos al final del array y no hayamos encontrado un día sin lluvias.

También es posible una solución sin utilizar la variable `boolean`:

```

1 public static int primerDiaSinLluvia2(double[] lluvia){
2 int i=0 ;
3 while (i<lluvia.length && lluvia[i] != 0)
4 i++;
5 if (i == lluvia.length) return -1 ;
6 else return i;
7 }

```

En este caso el subíndice `i` se incrementa mientras estemos dentro de los límites del array y no encontremos un día con lluvia `0`. Al finalizar el bucle hay que comprobar por cual de las dos razones finalizó: ¿Se encontró un día sin lluvias o se recorrió todo el array sin encontrar ninguno? En esta comprobación es importante no acceder al array si existe la posibilidad de que el subíndice esté fuera de los límites del array. La siguiente comprobación sería **incorrecta**:

```

1 if (lluvia[i] == 0) return i;
2 else return -1;

```

ya que, si se ha finalizado el bucle sin encontrar ningún día sin lluvia, `i` valdrá `lluvia.length`, que no es una posición válida del array, y al acceder a `lluvia[i]` se producirá la excepción `ArrayIndexOutOfBoundsException` (índice del array fuera de los límites)

Por otra parte, el mismo problema se puede resolver utilizando la sentencia `for`, como hemos hecho otras veces. Sin embargo la solución parece menos intuitiva porque el cuerpo del `for` quedaría vacío:

```

1 public static int primerDiaSinLluvia3(double[] lluvia){
2 int i;
3 for (i=0; i<lluvia.length && lluvia[i] != 0; i++) /*Nada*/ ;
4 if (i == lluvia.length) return -1 ;
5 else return i;
6 }

```

Otra opción más:

```

1 public static int primerDiaSinLluvia4(double[] lluvia){
2 int i=0 ;
3 while (i<lluvia.length){
4 if (lluvia[i++] == 0) return i ;
5 }
6 return -1 ;
7 }

```

#### 12.3.2.2. BÚSQUEDA DESCENDENTE

En los ejemplos de búsqueda anteriores hemos iniciado la búsqueda en el elemento cero y hemos ido ascendiendo hasta la última posición del array. A esto se le llama búsqueda ascendente.

Si queremos encontrar el último día del mes en que no llovió podemos realizar una búsqueda descendente, es decir, partiendo del último componente del array y decremetando progresivamente el subíndice hasta llegar a la posición cero o hasta encontrar lo buscado:

```

1 public static int ultimoDiaSinLluvia(double[] lluvia){
2 int i=lluvia.length-1;
3 boolean encontrado = false ;
4 while (i>=0 && !encontrado){
5 if (lluvia[i] == 0) encontrado = true ;
6 else i-- ;
7 }
8 if (encontrado) return i ;
9 else return -1 ;
10 }

```

### 12.3.2.3. BÚSQUEDA EN UN ARRAY ORDENADO: BÚSQUEDA BINARIA

Suponga que una amiga apunta un número entre el 0 y el 99 en una hoja de papel y vosotros debéis adivinarlo. Cada vez que conteste, le dirá si el valor que ha dicho es mayor o menor que el que ha de adivinar. ¿Qué estrategia seguiría para lograrlo? Hay que pensar un algoritmo a seguir para resolver este problema.

Una aproximación muy ingenua podría ser ir diciendo todos los valores uno por uno, empezando por 0. Está claro que cuando llegue al 99 lo habréis adivinado. En el mejor caso, si había escrito el 0, acertará en la primera, mientras que en el peor caso, si había escrito el 99, necesitaríais 100 intentos. Si estaba por medio, tal vez con 40-70 basta. Este sería un algoritmo eficaz (hace lo que tiene que hacer), pero no muy eficiente (lo hace de la mejor manera posible). Ir probando valores al azar en lugar de hacer esto tampoco mejora gran cosa el proceso, y viene a ser lo mismo.

Si alguna vez habeis jugado a este juego, lo que habreis hecho es ser un poco más astutos y empezar por algún valor del medio. En este caso, por ejemplo, podría ser el 50. Entonces, en caso de fallar, una vez está seguro de si el valor secreto es mayor o menor que su respuesta, en el intento siguiente probar un valor más alto o más bajo , e ir haciendo esto repetidas veces.

Generalmente, la mejor estrategia para adivinar un número secreto entre 0 y N sería primer probar  $N/2$ . Si no se ha acertado, entonces si el número secreto es más alto se intenta adivinar entre  $(N/2 + 1)$  y N. Si era más bajo, se intenta adivinar el valor entre 0 y  $N-1$ . Para cada caso, se vuelve a probar el valor que hay en el medio del nuevo intervalo. Y así sucesivamente, haciendo cada vez más pequeño el intervalo de búsqueda, hasta adivinarlo. En el caso de 100 valores, esto garantiza que, en el peor de los casos, en 7 intentos seguro que se adivina. Esto es una mejora muy grande respecto al primer algoritmo, donde hacían falta 100 intentos, y por tanto, este sería un algoritmo más eficiente. Concretamente, siempre se adivinará en  $\log_2(N)$  intentos como máximo.

Si os fijáis, el ejemplo que se acaba de explicar, en realidad, no es más que un esquema de búsqueda en una secuencia de valores, como puede ser dentro de un array, partiendo de la condición que todos los elementos estén ordenados de menor a mayor. De hecho, hasta ahora, para hacer una búsqueda de un valor dentro de un array se ha usado el sistema "ingenuo", mirando una por una todas las posiciones. Pero si los elementos están ordenados previamente, se podría usar el sistema "astuto" para diseñar un algoritmo mucho más eficiente, y hasta cierto punto, más "inteligente".

El algoritmo basado en esta estrategia se conoce como **búsqueda binaria o dicotómica**.

Para ello iniciaremos la búsqueda en la posición central del array.

- Si el elemento central es el buscado habremos finalizado la búsqueda.
- Si el elemento central es mayor que el buscado, tendremos que continuar la búsqueda en la mitad izquierda del array ya que, al estar éste ordenado todos los elementos de la mitad derecha serán también mayores que el buscado.
- Si el elemento central es menor que el buscado, tendremos que continuar la búsqueda en la mitad derecha del array ya que, al estar éste ordenado todos los elementos de la mitad izquierda serán también menores que el buscado.

En un solo paso hemos descartado la mitad de los elementos del array. Para buscar en la mitad izquierda o en la mitad derecha utilizaremos el mismo criterio, es decir, iniciaremos la búsqueda en el elemento central de dicha mitad, y así sucesivamente hasta encontrar lo buscado o hasta que descubramos que no está.

Supongamos por ejemplo que, dado un array que contiene edades de personas, ordenadas de menor a mayor queremos averiguar si hay alguna persona de 36 años o no.

El siguiente método soluciona este problema realizando una búsqueda binaria:

```

1 public static boolean hayAlguienDe36(int[] edad) {
2 // Las variables izq y der marcarán el fragmento del array en el que
3 // realizamos la búsqueda. Inicialmente buscamos en todo el array.
4 final int NUMERO_BUSCADO = 36;
5 int izq = 0;
6 int der = edad.length - 1;
7 boolean encontrado = false;
8 while (izq <= der && !encontrado) {
9 // Calculamos posición central del fragmento en el que buscamos
10 int m = (izq + der) / 2;
11 if (edad[m] == NUMERO_BUSCADO) // Hemos encontrado una persona de 36
12 {
13 encontrado = true;
14 } else if (edad[m] > NUMERO_BUSCADO) {
15 // El elemento central tiene más de 36.
16 // Continuamos la búsqueda en la mitad izquierda. Es decir,
17 // entre las posiciones izq y m-1
18 der = m - 1;
19 } else {
20 // El elemento central tiene menos de 36.
21 // Continuamos la búsqueda en la mitad derecha. Es decir,
22 // entre las posiciones m+1 y der
23 izq = m + 1;
24 } // del if
25 } // del while
26 return encontrado; // if (encontrado) return true; else return false;
27 }

```

La búsqueda finaliza cuando encontramos una persona con 36 años (`encontrado==true`) o cuando ya no es posible encontrarla, circunstancia que se produce cuando `izq` y `der` se cruzan (`izq>der`).

### 12.3.3. Problemas de ordenación

Con frecuencia necesitamos que los elementos de un array estén ordenados (por ejemplo para usar la búsqueda binaria).

Existen multitud de algoritmos que permiten ordenar los elementos de un array, entre los que hay soluciones **iterativas** y soluciones **recursivas**.

Entre los algoritmos **iterativos** tenemos, por ejemplo, el **método de la burbuja**, el **método de selección directa** y el **método de inserción directa**.

Entre los **recursivos**, son conocidos el algoritmo **mergesort** y el **quickSort**, que realizan la ordenación más rápidamente que los algoritmos iterativos que hemos nombrado.

Como ejemplo vamos a ver como se realiza la ordenación de un array de enteros utilizando el método de **selección directa**:

```

1 public static void seleccionDirecta(int[] v) {
2 for (int i = 0; i < v.length-1; i++) {
3 //Localizamos elemento que tiene que ir en la posición i
4 int posMin = i;
5 //buscar el menor a la derecha
6 for (int j = i + 1; j < v.length; j++) {
7 if (v[j] < v[posMin]) {
8 posMin = j;
9 }
10 }
11 //al llegar aquí posMin tendrá la posición del elemento menor
12 //Intercambiamos los elementos de las posiciones i y posMin
13 //v[i]<=>v[posMin];
14 int aux = v[posMin];
15 v[posMin] = v[i];
16 v[i] = aux;
17 }
18 }

```

El método consiste en recorrer el array ascendente a partir de la posición cero.

En cada posición (`i`) localizamos el elemento que tiene que ocupar dicha posición cuando el array esté ordenado, es decir, el menor de los elementos que quedan a su derecha.

Cuando se ha determinado el menor se coloca en su posición realizando un intercambio con el elemento de la posición `i`. Con ello, el array queda ordenado hasta la posición `i`.

Y a modo de curiosidad os dejo por aquí el método de inserción directa:

1. Comenzamos considerando el primer elemento como la parte ordenada.
2. Luego, tomamos un elemento de la parte no ordenada y lo insertamos en la posición correcta dentro de la parte ordenada, desplazando los elementos mayores que él hacia la derecha.

3. Repetimos este proceso hasta que todos los elementos estén en la parte ordenada.

```

1 public static void insercionDirecta(int[] array) {
2 for (int i = 1; i < array.length; i++) {
3 int key = array[i];
4 int j = i - 1;
5
6 // Mover los elementos mayores que key hacia la derecha
7 while (j >= 0 && array[j] > key) {
8 array[j + 1] = array[j];
9 j--;
10 }
11
12 // Insertar key en su posición correcta
13 array[j + 1] = key;
14 }
15 }
```

### Más información

Ejemplos visuales de distintos métodos de ordenación, con distintos tipos de entradas: <https://www.toptal.com/developers/sorting-algorithms> Otro ejemplo de visualizador de algoritmos: <http://algorithm-visualizer.org/>

## 12.4. Arrays bidimensionales: matrices

Los arrays bidimensionales, también llamados matrices, son muy similares a los arrays que hemos visto hasta ahora: También son una colección de elementos del mismo tipo que se agrupan bajo un mismo nombre de variable. Sin embargo:

- Sus elementos están organizados en filas y columnas. Tienen, por tanto una altura y una anchura, y por ello se les llama bidimensionales.
- A cada componente de una matriz se accede mediante dos subíndices: el primero se refiere al número de fila y el segundo al número de columna. En la siguiente figura, `m[0][0]` es 2, `m[0][3]` es 9, `m[2][0]` es 57

|      |   | columna |    |    |    |
|------|---|---------|----|----|----|
|      |   | 0       | 1  | 2  | 3  |
| fila | 0 | 2       | 5  | 6  | 9  |
|      | 1 | 3       | 2  | 2  | 8  |
|      | 2 | 57      | 12 | 15 | 36 |
|      | 3 | 33      | 6  | 12 | 6  |
|      | 4 | 0       | 41 | 5  | 7  |

- Como vemos, filas y columnas se numeran a partir del 0.

Si se quisiera extender el tratamiento el estudio de la pluviosidad, para abarcar no solo los días de un mes sino los de todo un año, se podría definir, por ejemplo, un array de 366 elementos, que mantuviera de forma correlativa los datos de pluviosidad de una zona día a día. Con ello, por ejemplo, el dato correspondiente al día 3 de febrero ocuparía la posición 34 del array, mientras que el correspondiente al 2 de julio ocuparía el 184.

Una aproximación más conveniente para la representación de estos datos consistiría en utilizar una matriz con 12 filas (una por mes) y 31 columnas (una por cada día del mes). Esto permitiría una descripción más ajustada a la realidad y, sobre todo, simplificaría los cálculos de la posición real de cada día en la estructura de datos. El elemento `[0][3]` correspondería, por ejemplo, a las lluvias del 4 de enero.

### 12.4.1. Matrices en Java

#### Definición

En Java, una matriz es, en realidad un array en el que cada componente es, a su vez, un array. Dicho de otra manera, una matriz de enteros es un array de arrays de enteros.

Esto, que no es igual en otros lenguajes de programación, tiene ciertas consecuencias en la declaración, creación y uso de las matrices en Java:

- Una matriz, en Java, puede tener distinto número de elementos en cada fila.
- La creación de la matriz se puede hacer en un solo paso o fila por fila.
- Si `m` es una matriz de enteros...
- `m[i][j]` es el entero de la fila `i`, columna `j`
- `m[i]` es un array de enteros.
- `m.length` es el número de filas de `m`.
- `m[i].length` es el número de columnas de la fila `i`
- Podríamos dibujar la matriz `m` del ejemplo anterior de una forma más cercana a cómo Java las representa internamente:



### 12.4.2. Declaración de matrices.

El código siguiente declara una matriz (array bidimensional) de elementos de tipo `double`, y la crea para que tenga 5 filas y 4 columnas (matriz de 5x4):

```
1 double[][] m1 = new double[5][4];
```

La siguiente declaración es equivalente a la anterior aunque en la práctica es menos utilizada a no ser que queramos que cada fila tenga un número distinto de elementos:

```
1 double[][] m2 = new double[5][];
2 m2[0] = new double[4];
3 m2[1] = new double[4];
4 m2[2] = new double[4];
5 m2[3] = new double[4];
6 m2[4] = new double[4];
```

Es posible inicializar cada uno de los subarrays con un tamaño diferente (aunque el tipo base elemental debe ser siempre el mismo para todos los componentes). Por ejemplo:

```
1 double[][] m3 = new double[5][];
2 m3[0] = new double[3];
3 m3[1] = new double[4];
4 m3[2] = new double[14];
5 m3[3] = new double[10];
6 m3[4] = new double[9];
```

### 12.4.3. Inicialización.

La forma de inicializar una matriz de enteros de por ejemplo [4][3] seria:

```

1 int[][] m4 = {{7,2,4},{8,2,5},{9,4,3},{1,2,4}};
2
3 //aunque se entiende mejor de este modo:
4 int[][] m4 = {
5 {7,2,4},
6 {8,2,5},
7 {9,4,3},
8 {1,2,4}
9 };

```

|   |   |   |
|---|---|---|
| 7 | 2 | 4 |
| 8 | 2 | 5 |
| 9 | 4 | 3 |
| 1 | 2 | 4 |

### 12.4.4. Recorrido

El recorrido se hace de forma similar al de un array aunque, dado que hay dos subíndices, será necesario utilizar dos bucles anidados: uno que se ocupe de recorrer las filas y otro que se ocupe de recorrer las columnas.

El siguiente fragmento de código recorre una matriz `m4` para imprimir sus elementos uno a uno.

```

1 //recorrido por filas
2 System.out.println("\nRecorrido por filas: ");
3 for (int f = 0; f < m4.length; f++) {
4 for (int c = 0; c < m4[f].length; c++) {
5 System.out.print(m4[f][c] + " ");
6 }
7 System.out.println("");
8 }
9 //Recorrido por filas:
10 //7 2 4
11 //8 2 5
12 //9 4 3
13 //1 2 4

```

El recorrido se ha hecho por filas, es decir, se imprimen todos los elementos de una fila y luego se pasa a la siguiente. Como habíamos indicado anteriormente, `m.length` representa el número de filas de `m`, mientras que `m[i].length` el número de columnas de la fila `i`.

También es posible hacer el recorrido por columnas: imprimir la columna 0, luego la 1, etc:

```

1 System.out.println("\nRecorrido por columnas: ");
2 int numFilas = m4.length;
3 int numColumnas = m4[0].length;
4 for (int c = 0; c < numColumnas; c++) {
5 for (int f = 0; f < numFilas; f++) {
6 System.out.print(m4[f][c] + " ");
7 }
8 System.out.println("");
9 }
10 //Recorrido por columnas:
11 //7 8 9 1
12 //2 2 4 2
13 //4 5 3 4

```

o, directamente ...

```

1 System.out.println("\nRecorrido por columnas versión 2: ");
2 for (int c = 0; c < m4[0].length; c++) {
3 for (int f = 0; f < m4.length; f++) {
4 System.out.print(m4[f][c] + " ");
5 }
6 System.out.println("");
7 }
8 //Recorrido por columnas versión 2:
9 //7 8 9 1
10 //2 2 4 2
11 //4 5 3 4

```

En este caso, para un funcionamiento correcto del recorrido sería necesario que todas las columnas tuvieran igual número de elementos, pues en el bucle externo, se toma como referencia para el número de columnas la longitud de `m[0]`, es decir el número de elementos de la primera fila.

## 12.5. Arrays multidimensionales

En el punto anterior hemos visto que podemos definir arrays cuyos elementos son a la vez arrays, obteniendo una estructura de datos a la que se accede mediante dos subíndices, que hemos llamado arrays bidimensionales o matrices.

Este *anidamiento* de estructuras se puede generalizar, de forma que podríamos construir arrays de más de dos dimensiones. En realidad Java no pone límite al número de subíndices de un array. Podríamos hacer declaraciones como las siguientes:

```
1 int[][][] notas = new int[10][5][3]; //Notas de 10 alum. en 5 asign. en 3 eval.
2 notas[2][3][1]=5; //El alumno 2, para la asignatura 3 de la primera evaluación ha sacado un 5
3 double[][][][][] w = new double [2][7][10][4][10];
```

### Más información

Sin embargo, encontrar ejemplos en los que sean necesarios arrays de más de tres dimensiones es bastante raro, y aún cuando los encontramos solemos utilizar arrays de uno o dos subíndices porque nos resulta menos complejo manejarlos.

## 12.6. Recursividad

# Entendiendo la Recursividad

Una técnica de programación poderosa donde un método se invoca a sí mismo para resolver problemas complejos dividiéndolos en versiones más simples hasta alcanzar una solución directa.

### Fundamentos de la Recursividad

#### Un método que se invoca a sí mismo

Resuelve un problema complejo dividiéndolo en versiones más simples de sí mismo.

#### El Caso Base: La condición de salida

Es la condición donde el problema es tan simple que se resuelve directamente, deteniendo las llamadas.

#### El Caso Recursivo: La llamada a sí mismo

Llama al mismo método, pero con datos que se acercan progresivamente al caso base.

### Aplicación y Precauciones

#### Ejemplo Clásico: Cálculo del Factorial (n!)

La definición recursiva es  $n! = n * (n-1)!$ , con un caso base de  $0! = 1$ .

| Operación                                                   | Retorno           |
|-------------------------------------------------------------|-------------------|
| Esperando...                                                | Esperando...      |
| Retorno                                                     | Esperando...      |
| Retorno                                                     | Esperando...      |
| Retorno                                                     | Esperando...      |
| Llamada factorial(3)<br>Operación $3 * \text{factorial}(2)$ |                   |
| Llamada factorial(2)<br>Operación $2 * \text{factorial}(1)$ |                   |
| Llamada factorial(1)<br>Operación $1 * \text{factorial}(0)$ |                   |
| Llamada factorial(0)<br>Operación Caso Base                 | Retorno Retorna 1 |



#### Riesgo Principal: Desbordamiento de Pila (Stack Overflow)

Ocurre si el caso base nunca se alcanza, agotando la memoria con llamadas infinitas.

NotebookLM

A la hora de crear programas complejos, uno de los aspectos que diferencia el buen programador del aficionado es su capacidad de hacer algoritmos eficientes. O sea, que sean capaces de resolver el problema planteado en el mínimo de pasos. En el caso de un programa, esto significa la necesidad de ejecutar el mínimo número de instrucciones posible. Ciertamente, si el resultado tiene que ser exactamente el mismo, siempre será mejor hacer una tarea en 10 pasos que en 20, intentando evitar pasos que en realidad son innecesarios. Por lo tanto, la etapa de diseño de un algoritmo es bastante importante y hay que pensar bien una estrategia eficiente. Ahora bien, normalmente, los algoritmos más eficientes también son más difíciles de pensar y codificar, ya que no siempre son evidentes.

### 12.6.1. Aplicación de la recursividad

A menudo encontrareis que explicar de palabra la idea general de una estrategia puede ser sencillo, pero traducirla instrucciones de Java ya no lo es tanto. Retomamos ahora el caso de la búsqueda dicotómica o binaria, dado que hay que ir repitiendo unos pasos en sucesivas iteraciones, está más o menos claro que el problema planteado para realizar búsquedas eficientes se basa en una estructura de repetición. Pero no se recorren todos los elementos y el índice no se incrementa uno a uno, sino que se va cambiando a valores muy diferentes para cada iteración. No es un caso evidente. Precisamente, este ejemplo no se ha elegido al azar, ya que es un caso en el que os puede ir bien aplicar un nuevo concepto que permite facilitar la definición de algoritmos complejos donde hay repeticiones.

#### Definición

La **recursividad** es una forma de describir un proceso para resolver un problema de manera que, a lo largo de esta descripción, se usa el proceso mismo que se está describiendo, pero aplicado a un caso más simple.

De hecho, tal vez sin darse cuenta de ello en, ya se ha usado recursividad para describir cómo resolver un problema. Para ver qué significa exactamente la definición formal apenas descrita, se repetirá el texto en cuestión, pero remarcando el aspecto recursivo de la descripción:

*"Generalmente, la mejor estrategia para adivinar un número secreto entre 0 y N sería primero probar N/2. Si no se ha acertado, entonces si el número secreto es más alto se intenta adivinar entre (N/2 + 1) y N. Si era más bajo, se intenta adivinar el valor entre 0 y N-1. Para cada caso, se vuelve a probar el valor que hay en el centro del nuevo intervalo. Y así sucesivamente, hasta adivinarlo."*

O sea, **el proceso de adivinar un número se basa en el proceso de intentar adivinar un número!** Esto parece hacer trampas, ya es como usar la misma palabra que se quiere definir a su propia definición. Pero fíjese en un detalle muy importante. Los nuevos usos del proceso de "adivinar" son casos más simples, ya que primero se adivina entre N valores posibles, luego entre N/2 valores, después entre N/4, etc. Este hecho no es casual y de él depende poder definir un proceso recursivo de manera correcta.

#### Más ejemplos

Otro ejemplo de recursividad es la definición de las iniciales del sistema operativo GNU quieren decir "GNU is Not Unix"

### 12.6.2. Implementación de la recursividad

La implementación de la recursividad dentro del código fuente de un programa se hace a nivel de método.

#### Definición

Un **método recursivo** es aquel que, dentro de su bloque de instrucciones, tiene alguna invocación a él mismo.

El bloque de código de un método recursivo siempre se basa en una estructura de selección múltiple, donde cada rama es de alguno de los dos casos posibles descritos a continuación.

- Por un lado, en el **caso base**, que contiene un bloque instrucciones dentro de las cuales no hay ninguna llamada al método mismo. Se ejecuta cuando se considera que, a partir de los parámetros de entrada, el problema ya es suficientemente simple como para ser resuelto directamente. En el caso de la búsqueda, sería cuando la posición intermedia es exactamente el valor que se está buscando, o bien cuando ya se puede decidir que el elemento a buscar no existe.
- Por otra parte, existe el **caso recursivo**, que contiene un bloque de instrucciones dentro de las cuales hay una llamada al método mismo, dado que se considera que aún no se puede resolver el problema fácilmente. Ahora bien, los valores usados como parámetros de esta nueva llamada deben ser diferentes a los originales. Concretamente, serán unos valores que tiendan a acercarse al caso base. En el caso de la búsqueda, se corresponde a la búsqueda sobre la mitad de los valores originales, ya sea hacia la mitad inferior o superior.

Este es un caso en el que el intervalo de posiciones donde se hará la nueva búsqueda se va acercando al caso base, ya que tarde o temprano, llamada tras llamada, el espacio de búsqueda se irá reduciendo hasta que, o bien se encuentra el elemento, o queda claro que no está.

Dentro de la estructura de selección siempre debe haber al menos un caso base y uno recursivo. Normalmente, los algoritmos recursivos más sencillos tienen uno de cada. Es imprescindible que los casos recursivos siempre garanticen que sucesivas llamadas van aproximando los valores de los parámetros de entrada a algún caso base, ya que, de lo contrario, el programa nunca termina y se produce el mismo efecto que en un bucle infinito.

#### 12.6.2.1. CÁLCULO RECURSIVO DE LA OPERACIÓN FACTORIAL

Como ejemplo del funcionamiento de un método recursivo, se empezará con un caso sencillo. Se trata del cálculo de la llamada operación **factorial** de un valor entero positivo. Esta es unaria y se expresa con el operador exclamación (por ejemplo,  $4! = 1 * 2 * 3 * 4$ ,  $20! = 1 * 2 * 3 * \dots * 20$ ,  $3! = 1 * 2 * 3$ ). El resultado de esta operación es la multiplicación de todos los valores desde el 1 hasta el indicado ( $7! = 1 * 2 * 3 * 4 * 5 * 6 * 7$ ). Normalmente, la definición matemática de esta operación se hace de manera recursiva:

- $0! = 1$  ↪ **caso base**
- $n! = n * (n - 1)!$  ↪ **caso recursivo**

Así pues, tened en cuenta que el caso recursivo realiza un cálculo que depende de usar la propia definición de la operación, pero cuando lo hace es con un nuevo valor inferior al original, por lo que se garantiza que, en algún momento, se hará una llamada recursiva que desembocará en el caso base. Cuando esto ocurra, la cadena de llamadas recursivas acabará. Una manera de ver esto es desarrollando paso a paso esta definición:

```

1 4! = 4 * (4 - 1)! = 4 * (3)!
2 4 * 3! = 4 * (3 * (3-1)!) = 4 * 3 * (2)!
3 4 * 3 * 2! = 4 * 3 * (2 * (2-1)!) = 4 * 3 * 2 * (1)!
4 4 * 3 * 2 * 1! = 4 * 3 * 2 * (1 * (1 - 1)!) = 4 * 3 * 2 * 1 * (0)!
5 4 * 3 * 2 * 1 * 0! = 4 * 3 * 2 * 1 * (1) = 24

```

Su implementación en Java sería la que ves más abajo. Ahora bien, en este código se han añadido algunas sentencias para escribir información por pantalla, de forma que se vea con más detalle cómo funciona un método recursivo. Veréis que, inicialmente, se llevan a cabo una serie de invocaciones del caso recursivo, uno tras otro, hasta que se llega a una llamada que ejecuta el caso base. Es a partir de entonces cuando, a medida que se van ejecutando las sentencias `return` del caso recursivo, realmente se va acumulando el cálculo. Otra forma de verlo es depurando el programa.

```

1 package UD04;
2
3 public class Recursividad {
4
5 public static void main(String[] args) {
6 //factorial
7 System.out.println(factorial(4));
8
9 [...]
10 }
11
12 /**
13 * Método recursivo que calcula el factorial
14 */
15 public static int factorial(int n) {
16 if (n == 0) {
17 //Caso base: Se sabe el resultado directamente
18 System.out.println("Caso base: n es igual a 0");
19 return 1;
20 } else {
21 //Caso recursivo: Para calcularlo hay que invocar al método recursivo
22 //El valor del nuevo parámetro de entrada se ha de modificar, de
23 //manera que se vaya acercando al caso base
24 System.out.println("Caso recursivo " + (n - 1)
25 + ": Se invoca al factorial(" + (n - 1) + ")");
26 int res = n * factorial(n - 1);
27 System.out.println(" cuyo resultado es: " + res);
28 return res;
29 }
30 }
31 [...]

```

La ejecución resultante es:

```

1 Caso recursivo 3: Se invoca al factorial(3)
2 Caso recursivo 2: Se invoca al factorial(2)
3 Caso recursivo 1: Se invoca al factorial(1)
4 Caso recursivo 0: Se invoca al factorial(0)
5 Caso base: n es igual a 0
6 cuyo resultado es: 1
7 cuyo resultado es: 2
8 cuyo resultado es: 6
9 cuyo resultado es: 24
10 24

```

#### 12.6.2.2. CÁLCULO RECURSIVO DE LA BÚSQUEDA DICOTÓMICA

A continuación se muestra el código del algoritmo recursivo de búsqueda dicotómica o binaria sobre un array. Observad atentamente los comentarios, los cuales identifican los casos base y recursivos. En este caso, hay más de un caso base y recursivo.

```

1 package UD04;
2
3 public class Recursividad {
4
5 public static void main(String[] args) {
6 [...]
7 //busqueda binaria recursiva
8 int[] array = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20};
9 int buscaDieciocho = BusquedaBinaria(array, 0, array.length - 1, 18);
10 int buscaCinco = BusquedaBinaria(array, 0, array.length - 1, 5);
11 System.out.println("Busqueda del 18: " + buscaDieciocho);
12 System.out.println("Busqueda del 5: " + buscaCinco);
13 [...]
14 }
15
16 [...]
17 public static int BusquedaBinaria(int[] array, int inicio, int fin, int valor) {
18 if (inicio > fin) {
19 //Caso base: No se ha encontrado el valor
20 return -1;
21 }
22 //Se calcula la posición central entre los dos indices de búsqueda
23 int pos = inicio + (fin - inicio) / 2;
24 if (array[pos] > valor) {
25 //Caso recursivo: Si el valor es menor que la posición que se ha
26 //consultado, entonces hay que seguir buscando por la parte
27 //''derecha'' del array
28 return BusquedaBinaria(array, inicio, pos - 1, valor);
29 } else if (array[pos] < valor) {
30 //Caso recursivo: Si el valor es mayor que la posición que se ha
31 //consultado, entonces hay que seguir buscando por la parte
32 //''izquierda'' del array
33 return BusquedaBinaria(array, pos + 1, fin, valor);
34 } else {
35 //caso base: Es igual, por tanto, se ha encontrado
36 return pos;
37 }
38 }
39 [...]
40 }

```

El resultado de la ejecución es:

```

1 Busqueda del 18: 8
2 Busqueda del 5: -1

```

Prácticamente cualquier problema que se puede resolver con un algoritmo recursivo también se puede resolver con sentencias de estructuras de repetición (de manera iterativa). Pero muy a menudo su implementación será mucho menos evidente y las interacciones entre instrucciones bastante más complejas que la opción recursiva (una vez se entiende este concepto, claro).

### 12.6.3. Desbordamiento de pila (stack overflow)

Las versiones recursivas de muchas rutinas pueden ejecutarse un poco más lentamente que sus equivalentes iterativos debido a la sobrecarga adicional de las llamadas a métodos adicionales. Demasiadas llamadas recursivas a un método podrían causar un **desbordamiento de la pila**.

Como el almacenamiento para los parámetros y las variables locales está en la pila y cada llamada nueva crea una nueva copia de estas variables, es posible que la pila se haya agotado. Si esto ocurre, el sistema de tiempo de ejecución (run-time) de Java causará una excepción. Sin embargo, probablemente no tendrás que preocuparte por esto a menos que una rutina recursiva se vuelva loca.

La principal ventaja de la recursividad es que algunos tipos de algoritmos se pueden implementar de forma más clara y más recursiva de lo que pueden ser iterativamente. Por ejemplo, el algoritmo de clasificación [Quicksort](#) es bastante difícil de implementar de forma iterativa. Además, algunos problemas, especialmente los relacionados con la **IA**, parecen prestarse a **soluciones recursivas**.

```

1 package UD04;
2
3 public class Recursividad {
4
5 public static void main(String[] args) {
6 [...]
7 //desbordamiento de pila
8 desbordamientoPila(10);
9 }
10 [...]
12
13 public static int desbordamientoPila(int n) {
14 // condición base incorrecta (esto provoca un desbordamiento de la pila).
15 if (n == 100) {
16 return 1;
17 } else {
18 return n * desbordamientoPila(n - 1);
19 }
20 }
21 }
```

En el ejemplo anterior si se llama a `desbordamientoPila(10)`, llamará a `desbordamientoPila(9)`, `desbordamientoPila(8)`, `desbordamientoPila(7)`, etc., pero el número nunca llegará a 100. Por lo tanto, no se alcanza la condición base. Si la memoria se agota con estos métodos en la pila, provocará un error de desbordamiento de pila (`java.lang.StackOverflowError`).



### Importante

Al escribir métodos recursivos, debe tener una instrucción condicional, como un `if`, en algún lugar para forzar el retorno del método sin que se ejecute la llamada recursiva. Si no lo hace, una vez que llame al método, nunca retornará. Este tipo de error es muy común cuando se trabaja con recursividad.

## 12.7. Ejemplos UD04

### 12.7.1. EjemploUD04

```

1 package UD04;
2
3 import java.util.Scanner;
4
5 public class EjemploUD04 {
6
7 public static void main(String[] args) {
8 //declaración
9 double lluvia1[]; // lluvia1 es un array de double
10 double[] lluvia2; // lluvia2 es un array de double
11
12 //instanciación
13 lluvia1 = new double[31];
14
15 //declaración + instanciación
16 double lluvia3[] = new double[31];
17
18 // usamos un array para almacenar las edades de un grupo de personas
19 // la variable numPersonas contiene el número de personas del grupo
20 // y se asigna en tiempo de ejecución
21 Scanner teclado = new Scanner(System.in);
22 System.out.print("Introduce cuantos elementos debe tener el array edad[]:");
23 int numPersonas = teclado.nextInt();
24 int edad[] = new int[numPersonas];
25
26 //acceso a componentes
27 System.out.print("Introduce el dato para el componente 0: ");
28 edad[0] = teclado.nextInt();
29 System.out.println("El componente [0] vale " + edad[0]);
30 edad[1] = edad[0] + 1;
31 edad[2] = edad[0] + edad[1];
32 edad[2]++;
33 System.out.println("El componente [1] vale " + edad[1]);
34 System.out.println("El componente [2] vale " + edad[2]);
35
36 //el indice también admite calculos/variables:
37 int i = 3;
38 edad[i] = edad[i + 1];
39 edad[i + 2] = edad[i];
40
41 //Inicialización
42 int edad2[] = new int[10];
43 edad2[0] = 25;
44 edad2[1] = 10;
45 edad2[2] = 12;
46 //...
47
48 int edad3[] = {25, 10, 23, 34, 65, 23, 1, 67, 54, 12};
49
50 //Ejemplo práctico
51 System.out.println(nombreMes(3)); //marzo
52
53 //Paso de arrays como parámetros:
54 int a = 1;
55 int v[] = {1, 1, 1};
56 metodo(v, a); //Pasar un array como parámetro
57 System.out.println(a); // Muestra 1
58 System.out.println(v[0]); // Muestra 2
59
60 //atributo lenght
61 double estatura[] = new double[25];
62 System.out.println(estatura.length); // Mostrará por pantalla: 25
63
64 //Array args[] del método main contiene los parámetros de entrada
65 System.out.println(args[0]); //parámetro 1 de la línea de comandos
66 System.out.println(args[1]); //parámetro 2 de la línea de comandos
67
68 //busquedas y recorridos de arrays
69 double pluviosidad[] = {5, 4, 0, 0, 10, 0, 0, 0, 0, 2, 2, 3, 4, 0,
70 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 6, 0, 0};
71
72 //recorrido ascendente
73 System.out.println(pluviosidadMediaAscendente(pluviosidad)); //1.3
74 //recorrido descendente
75 System.out.println(pluviosidadMediaDescendente(pluviosidad)); //1.3
76 //recorrido para máximo
77 System.out.println(pluviosidadMaxima(pluviosidad)); //10.0
78 //busqueda con while y boolean
79 System.out.println(primerDiaSinLluvia1(pluviosidad)); //2
80 //busqueda con while sin boolean
81 System.out.println(primerDiaSinLluvia2(pluviosidad)); //2
82 //busqueda con for
83 System.out.println(primerDiaSinLluvia3(pluviosidad)); //2
84 //busqueda descendente
85 System.out.println(ultimoDiaSinLluvia(pluviosidad)); //31

```

```

1 //busqueda en arrays ordenados (busqueda binaria)
2 int buscarEdad[] = {15, 22, 33, 36, 41, 56, 71, 92};
3 System.out.println(hayAlguienDe36(buscarEdad)); //true
4
5 //ordenar arrays
6 int desordenado[] = {62, 4, 25, 27, 32, 1, 80, 43, 22};
7 seleccionDirecta(desordenado);
8 //con el siguiente bucle recorremos el array ascendente y al
9 //imprimirlo resulta en: 1 4 22 25 27 32 43 62 80
10 for (int j = 0; j <= desordenado.length - 1; j++) {
11 System.out.print(desordenado[j] + " ");
12 }
13
14 //arrays bidimensionales
15 double m1[][] = new double[5][4];
16
17 //con el mismo número de columnas
18 double m2[][] = new double[5][];
19 m2[0] = new double[4];
20 m2[1] = new double[4];
21 m2[2] = new double[4];
22 m2[3] = new double[4];
23 m2[4] = new double[4];
24
25 //diferentes números de columnas
26 double m3[][] = new double[5][];
27 m3[0] = new double[3];
28 m3[1] = new double[4];
29 m3[2] = new double[14];
30 m3[3] = new double[10];
31 m3[4] = new double[9];
32
33 int m4[][] = {
34 {7, 2, 4},
35 {8, 2, 5},
36 {9, 4, 3},
37 {1, 2, 4}
38 };
39
40 //recorrido por filas
41 System.out.println("\nRecorrido por filas: ");
42 for (int r = 0; r < m4.length; r++) {
43 for (int s = 0; s < m4[r].length; s++) {
44 System.out.print(m4[r][s] + " ");
45 }
46 System.out.println("");
47 }
48 //Recorrido por filas:
49 //7 2 4
50 //8 2 5
51 //9 4 3
52 //1 2 4
53
54 System.out.println("\nRecorrido por columnas: ");
55 int numFilas = m4.length;
56 int numColumnas = m4[0].length;
57 for (int j = 0; j < numColumnas; j++) {
58 for (int k = 0; k < numFilas; k++) {
59 System.out.print(m4[k][j] + " ");
60 }
61 System.out.println("");
62 }
63 //Recorrido por columnas:
64 //7 8 9 1
65 //2 2 4 2
66 //4 5 3 4
67
68 System.out.println("\nRecorrido por columnas versión 2: ");
69 for (int j = 0; j < m4[0].length; j++) {
70 for (int k = 0; k < m4.length; k++) {
71 System.out.print(m4[k][j] + " ");
72 }
73 System.out.println("");
74 }
75 //Recorrido por columnas versión 2:
76 //7 8 9 1
77 //2 2 4 2
78 //4 5 3 4
79
80 //arrays multidimensionales:
81 int notas[][][] = new int[10][5][3]; //Notas de 10 alum. en 5 asign. en 3 eval.
82 double w[][][][][] = new double [2][7][10][4][10];
83 }
84
85 public static String nombreMes(int mes) {
86 String nombre[] = { "", "enero", "febrero", "marzo", "abril",
87 "mayo", "junio", "julio", "agosto", "septiembre", "octubre",
88 "noviembre", "diciembre"};
89 return nombre[mes];
90 }
```

```

1 public static void metodo(int x[], int y) { //recibir un array como parámetro
2 x[0]++;
3 y++;
4 }
5
6 //recorremos ascendenteamente el array para obtener la media
7 public static double pluviosidadMediaAscendente(double lluvia[]) {
8 double suma = 0;
9 //Recorremos el array
10 for (int i = 0; i < lluvia.length; i++) {
11 suma += lluvia[i];
12 }
13 double media = suma / lluvia.length;
14 return media;
15 }
16
17 //recorremos descendenteamente el array para obtener la media
18 public static double pluviosidadMediaDescendente(double lluvia[]) {
19 double suma = 0;
20 //Recorremos el array
21 for (int i = lluvia.length - 1; i >= 0; i--) {
22 suma += lluvia[i];
23 }
24 double media = suma / lluvia.length;
25 return media;
26 }
27
28 //recorremos el array para encontrar el dia con más pluviosidad
29 public static double pluviosidadMaxima(double lluvia[]) {
30 // Suponemos el la pluviosidad máxima se produjo el primer día
31 double max = lluvia[0];
32 //Recorremos el array desde la posición 1, comprobando si hay una pluviosidad mayor
33 for (int i = 1; i < lluvia.length; i++) {
34 if (lluvia[i] > max) {
35 max = lluvia[i];
36 }
37 }
38 return max;
39 }
40 //Devolveremos el subíndice del primer componente del array cuyo valor es cero.
41 // Si no hay ningún día sin lluvias devolveremos -1
42
43 public static int primerDiaSinLluvia1(double lluvia[]) {
44 int i = 0;
45 boolean encontrado = false;
46 while (i < lluvia.length && !encontrado) {
47 if (lluvia[i] == 0) {
48 encontrado = true;
49 } else {
50 i++;
51 }
52 }
53 if (encontrado) {
54 return i;
55 } else {
56 return -1;
57 }
58 }
59
60 public static int primerDiaSinLluvia2(double lluvia[]) {
61 int i = 0;
62 while (i < lluvia.length && lluvia[i] != 0) {
63 i++;
64 }
65 if (i == lluvia.length) {
66 return -1;
67 } else {
68 return i;
69 }
70 }
71
72 public static int primerDiaSinLluvia3(double lluvia[]) {
73 int i;
74 for (i = 0; i < lluvia.length && lluvia[i] != 0; i++) /*Nada*/ ;
75 if (i == lluvia.length) {
76 return -1;
77 } else {
78 return i;
79 }
80 }

```

```

1 public static int ultimoDiaSinLluvia(double lluvia[]) {
2 int i = lluvia.length - 1;
3 boolean encontrado = false;
4 while (i >= 0 && !encontrado) {
5 if (lluvia[i] == 0) {
6 encontrado = true;
7 } else {
8 i--;
9 }
10 }
11 if (encontrado) {
12 return i;
13 } else {
14 return -1;
15 }
16 }
17
18 public static boolean hayAlguienDe36(int edad[]) {
19 // Las variables izq y der marcarán el fragmento del array en el que
20 // realizamos la búsqueda. Inicialmente buscamos en todo el array.
21 int izq = 0;
22 int der = edad.length - 1;
23 boolean encontrado = false;
24 while (izq <= der && !encontrado) {
25 // Calculamos posición central del fragmento en el que buscamos
26 int m = (izq + der) / 2;
27 if (edad[m] == 36) // Hemos encontrado una persona de 36
28 {
29 encontrado = true;
30 } else if (edad[m] > 36) {
31 // El elemento central tiene más de 36.
32 // Continuamos la búsqueda en la mitad izquierda. Es decir,
33 // entre las posiciones izq y m-1
34 der = m - 1;
35 } else {
36 // El elemento central tiene menos de 36.
37 // Continuamos la búsqueda en la mitad derecha. Es decir,
38 // entre las posiciones m+1 y der
39 izq = m + 1;
40 } // del if
41 } // del while
42 return encontrado; // if (encontrado) return true; else return false;
43 }
44
45 public static void seleccionDirecta(int v[]) {
46 for (int i = 0; i < v.length - 1; i++) {
47 //Localizamos elemento que tiene que ir en la posición i
48 int posMin = i;
49 for (int j = i + 1; j < v.length; j++) {
50 if (v[j] < v[posMin]) {
51 posMin = j;
52 }
53 }
54 //Intercambiamos los elementos de las posiciones i y posMin
55 int aux = v[posMin];
56 v[posMin] = v[i];
57 v[i] = aux;
58 }
59 }
60 }
```

## 12.7.2. Recursividad

```

1 package UD04;
2
3 public class Recursividad {
4
5 public static void main(String[] args) {
6 //factorial
7 System.out.println(factorial(4));
8
9 //busqueda binaria recursiva
10 int[] array = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20};
11 int buscaDieciocho = BusquedaBinaria(array, 0, array.length - 1, 18);
12 int buscaCinco = BusquedaBinaria(array, 0, array.length - 1, 5);
13 System.out.println("Busqueda del 18: " + buscaDieciocho);
14 System.out.println("Busqueda del 5: " + buscaCinco);
15
16 //desbordamiento de pila
17 desbordamientoPila(9);
18 }
19
20 /**
21 * Método recursivo que calcula el factorial
22 */
23 public static int factorial(int n) {
24 if (n == 0) {
25 //Caso base: Se sabe el resultado directamente
26 System.out.println("Caso base: n es igual a 0");
27 return 1;
28 } else {
29 //Caso recursivo: Para calcularlo hay que invocar al método recursivo
30 //El valor del nuevo parámetro de entrada se ha de modificar, de
31 //manera que se vaya acercando al caso base
32 System.out.println("Caso recursivo " + (n - 1)
33 + ": Se invoca al factorial(" + (n - 1) + ")");
34 int res = n * factorial(n - 1);
35 System.out.println(" cuyo resultado es: " + res);
36 return res;
37 }
38 }
39
40 public static int BusquedaBinaria(int[] array, int inicio, int fin, int valor) {
41 if (inicio > fin) {
42 //Caso base: No se ha encontrado el valor
43 return -1;
44 }
45 //Es calcula la posición central entre los dos índices de cerca
46 int pos = inicio + (fin - inicio) / 2;
47 if (array[pos] > valor) {
48 //Caso recursivo: Si el valor es menor que la posición que se ha
49 //consultado, entonces hay que seguir buscando por la parte
50 //''derecha'' del array
51 return BusquedaBinaria(array, inicio, pos - 1, valor);
52 } else if (array[pos] < valor) {
53 //Caso recursivo: Si el valor es mayor que la posición que se ha
54 //consultado, entonces hay que seguir buscando por la parte
55 //''izquierda'' del array
56 return BusquedaBinaria(array, pos + 1, fin, valor);
57 } else {
58 //caso base: Es igual, por tanto, se ha encontrado
59 return pos;
60 }
61 }
62
63 public static int desbordamientoPila(int n) {
64 // condición base incorrecta (esto provoca un desbordamiento de la pila).
65 if (n == 100) {
66 return 1;
67 } else {
68 return n * desbordamientoPila(n - 1);
69 }
70 }
71 }
```

## 12.8. Píldoras informáticas relacionadas

⌚10 de enero de 2026

## 13. 5.2 Anexo Cheatsheet Strings en Java

### 13.1. Introducción

Desde el punto de vista de la programación diaria, uno de los tipos de datos más importantes de Java es **String**. *String* define y admite cadenas de caracteres. En algunos otros lenguajes de programación, una cadena o string es una matriz o array de caracteres. Este no es el caso con Java. En Java, **los String son objetos**.

En realidad, has estado usando la clase String desde el comienzo del curso, pero no lo sabías. Cuando crea un literal de cadena, en realidad está creando un objeto String. Por ejemplo, en la declaración:

```
1 System.out.println("En Java, los String son objetos");
```

La clase String es bastante grande, y solo veremos una pequeña parte aquí.

### 13.2. Construyendo String

Puede construir un *String* igual que construye cualquier otro tipo de objeto: utilizando new y llamando al constructor *String*. Por ejemplo:

```
1 String str = new String("Hola");
```

Esto crea un objeto *String* llamado *str* que contiene la cadena de caracteres "Hola". También puedes construir una *String* desde otro *String*. Por ejemplo:

```
1 String str = new String("Hola");
2 String str2 = new String(str);
```

Después de que esta secuencia se ejecuta, *str2* también contendrá la cadena de caracteres "Hola". Otra forma fácil de crear una cadena se muestra aquí:

```
1 String str = "Estoy aprendiendo sobre String en JavadesdeCero.;"
```

En este caso, *str* se inicializa en la secuencia de caracteres "Estoy aprendiendo sobre String en JavadesdeCero.". Una vez que haya creado un objeto *String*, puede usarlo en cualquier lugar que permita una cadena entrecomillada. Por ejemplo, puede usar un objeto *String* como argumento para *println()*, como se muestra en este ejemplo:

```
1 // Uso de String
2 class DemoString
3 {
4 public static void main(String args[])
5 {
6 //Declaración de String de diferentes maneras
7 String str1=new String("En Java, los String son objetos");
8 String str2=new String("Se construyen de varias maneras");
9 String str3=new String(str2);
10
11 System.out.println(str1);
12 System.out.println(str2);
13 System.out.println(str3);
14
15 }
16 }
```

La salida del programa se muestra a continuación:

```
1 En Java, los String son objetos
2 Se construyen de varias maneras
3 Se construyen de varias maneras
```

### 13.3. Operando con Métodos de la clase String

La clase String contiene varios métodos que operan en cadenas. Aquí se detallan todos los métodos:

- int *length()* : Devuelve la cantidad de caracteres del String.

```
1 "Javadesdecero.es".length(); // retorna 16
```

- `Char charAt(int i)`: Devuelve el carácter en el índice *i*.

```
1 System.out.println("Javadesdecero.es".charAt(3)); // retorna 'a'
```

- `String substring(int i)`: Devuelve la subcadena del *i*-ésimo carácter de índice al final.

```
1 "Javadesdecero.es".substring(4); // retorna desdecero.es
```

- `String substring(int i, int j)`: Devuelve la subcadena del índice *i* a *j-1*.

```
1 "Javadesdecero.es".substring(4, 9); // retorna desde
```

- `String concat(String str)`: Concatena la cadena especificada al final de esta cadena.

```
1 String s1 = "Java";
2 String s2 = "desdeCero;
3 String salida = s1.concat(s2); // retorna "JavadesdeCero"
```

- `int indexOf(String s)`: Devuelve el índice dentro de la cadena de la primera aparición de la cadena especificada.

```
1 String s = "Java desde Cero";
2 int salida = s.indexOf("Cero"); // retorna 11
```

- `int indexOf(String s, int i)`: Devuelve el índice dentro de la cadena de la primera aparición de la cadena especificada, comenzando en el índice especificado.

```
1 String s = "Java desde Cero";
2 int salida = s.indexOf('a', 2); // retorna 3
```

- `int lastIndexOf(int ch)`: Devuelve el índice dentro de la cadena de la última aparición de la cadena especificada.

```
1 String s = "Java desde Cero";
2 int salida = s.lastIndexOf('a'); // retorna 3
```

- `boolean equals(Object otroObjeto)`: Compara este String con el objeto especificado.

```
1 Boolean salida = "Java".equals("Java"); // retorna true
2 Boolean salida = "Java".equals("java"); // retorna false
```

- `boolean equalsIgnoreCase(String otroString)`: Compares string to another string, ignoring case considerations.

```
1 Boolean salida= "Java".equalsIgnoreCase("Java"); // retorna true
2 Boolean salida = "Java".equalsIgnoreCase("java"); // retorna true
```

- `int compareTo(String otroString)`: Compara dos cadenas lexicográficamente.

```
1 int salida = s1.compareTo(s2); // donde s1 y s2 son strings que se comparan
2 /*
3 Esto devuelve la diferencia s1-s2. Si:
4 salida < 0 // s1 es menor que s2
5 salida = 0 // s1 y s2 son iguales
6 salida > 0 // s1 es mayor que s2
7 */
```

- `int compareToIgnoreCase(String otroString)`: Compara dos cadenas lexicográficamente, ignorando las consideraciones *case*.

```
1 int salida = s1.compareToIgnoreCase(s2); // donde s1 y s2 son strings que se comparan
2 /*
3 Esto devuelve la diferencia s1-s2. Si:
4 salida < 0 // s1 es menor que s2
5 salida = 0 // s1 y s2 son iguales
6 salida > 0 // s1 es mayor que s2
7 */
```

### Más información

En este caso, no considerará el *case* de una letra (ignorará si está en mayúscula o minúscula).

- `String toLowerCase()` : Convierte todos los caracteres de String a minúsculas.

```
1 String palabra1 = "HoLa";
2 String palabra2 = palabra1.toLowerCase(); // retorna "hola"
```

- `String toUpperCase()` : Convierte todos los caracteres de String a mayúsculas.

```
1 String palabra1 = "HoLa";
2 String palabra2 = palabra1.toUpperCase(); // retorna "HOLA"
```

- `String trim()` : Devuelve la copia de la cadena, eliminando espacios en blanco en ambos extremos. No afecta los espacios en blanco en el medio.

```
1 String palabra1 = " Java desde Cero ";
2 String palabra2 = palabra1.trim(); // retorna "Java desde Cero"
```

- `String replace(char oldChar, char newChar)` : Devuelve una nueva cadena al reemplazar todas las ocurrencias de `oldChar` con `newChar`.

```
1 String palabra1 = "yavadesdecero";
2 String palabra2 = palabra1.replace('y', 'j'); //retorna javadesdecero
```

### Más información

`palabra1` sigue siendo `yavadesdecero` y `palabra2`, `javadesdecero`

- `String replaceAll(String regex, String replacement)` : devuelve una cadena que reemplaza toda la secuencia de caracteres que coinciden con la expresión regular `regex` por la cadena de reemplazo `replacement`.

```
1 String str = "Ejemplo con espacios en blanco y tabs";
2 String str2 = str.replaceAll("\\s", ""); //retorna Ejemploconespaciosenblancoytabs
```

### Más información

Otras expresiones regulares (entre otras muchísimas):

- `\w` Cualquier cosa que sea un carácter de palabra
  - `\W` Cualquier cosa que no sea un carácter de palabra (incluida la puntuación, etc.)
  - `\s` Cualquier cosa que sea un carácter de espacio (incluido el espacio, los caracteres de tabulación, etc.)
  - `\S` Cualquier cosa que no sea un carácter de espacio (incluidas letras y números, así como puntuación, etc.)
- Debe escapar de la barra invertida si desea que `\s` alcance el motor de expresiones regulares, lo que da como resultado `\\\s`).
- Más información sobre expresiones regulares en java: <https://www.vogella.com/tutorials/JavaRegularExpressions/article.html>

### 13.4. Ejemplo de todos los métodos de String

```

1 // Código Java para ilustrar diferentes constructores y métodos
2 // de la clase String.
3
4 class DemoMetodosString
5 {
6 public static void main (String[] args)
7 {
8 String s= "JavadesdeCero";
9 // o String s= new String ("JavadesdeCero");
10
11 // Devuelve la cantidad de caracteres en la Cadena.
12 System.out.println("String length = " + s.length());
13
14 // Devuelve el carácter en el índice i.
15 System.out.println("Character at 3rd position = "
16 + s.charAt(3));
17
18 // Devuelve la subcadena del carácter índice i-ésimo
19 // al final de la cadena
20 System.out.println("Substring " + s.substring(3));
21
22 // Devuelve la subcadena del índice i a j-1.
23 System.out.println("Substring = " + s.substring(2,5));
24
25 // Concatena string2 hasta el final de string1.
26 String s1 = "Java";
27 String s2 = "desdeCero";
28 System.out.println("String concatenado = " +
29 s1.concat(s2));
30
31 // Devuelve el índice dentro de la cadena de
32 // la primera aparición de la cadena especificada.
33 String s4 = "Java desde Cero";
34 System.out.println("Índice de Cero: " +
35 s4.indexOf("Cero"));
36
37 // Devuelve el índice dentro de la cadena de
38 // la primera aparición de la cadena especificada,
39 // comenzando en el índice especificado.
40 System.out.println("Índice de a = " +
41 s4.indexOf('a',3));
42
43 // Comprobando la igualdad de cadenas
44 Boolean out = "Java".equals("java");
45 System.out.println("Comprobando la igualdad: " + out);
46 out = "Java".equals("Java");
47 System.out.println("Comprobando la igualdad: " + out);
48
49 out = "Java".equalsIgnoreCase("jaVA ");
50 System.out.println("Comprobando la igualdad: " + out);
51
52 int out1 = s1.compareTo(s2);
53 System.out.println("Si s1 = s2: " + out1);
54
55 // Conversión de cases
56 String palabra1 = "JavadesdeCero";
57 System.out.println("Cambiando a minúsculas: " +
58 palabra1.toLowerCase());
59
60 // Conversión de cases
61 String palabra2 = "JavadesdeCero";
62 System.out.println("Cambiando a MAYÚSCULAS: " +
63 palabra1.toUpperCase());
64
65 // Recortando la palabra
66 String word4 = " JavadesdeCero ";
67 System.out.println("Recortando la palabra: " + word4.trim());
68
69 // Reemplazar caracteres
70 String str1 = "YavadesdeCero";
71 System.out.println("String Original: " + str1);
72 String str2 = "YavadesdeCero".replace('Y','J') ;
73 System.out.println("Reemplazando Y por J -> " + str2);
74
75 // Reemplazar todos los caracteres
76 String strAll = "Ejemplo con espacios en blanco y tabs";
77 System.out.println("String Original: " + strAll);
78 String strAll2 = strAll.replaceAll("\\s", " ");
79 System.out.println("Eliminando todos los espacios en blanco -> " + strAll2);
80 }
81 }
```

Salida:

```

1 String length = 13
2 Character at 3rd position = a
3 Substring adesdeCero
4 Substring = vad
5 String concatenado = JavadesdeCero
6 Índice de Cero: 11
7 Índice de a = 3
8 Comprobando la igualdad: false
9 Comprobando la igualdad: true
10 Comprobando la igualdad: false
11 Si s1 = s2: -26
12 Cambiando a minúsculas: javadesdecero
13 Cambiando a MAYÚSCULAS: JAVADESCECERO
14 Recortando la palabra: JavadesdeCero
15 String Original: YavadesdeCero
16 Reemplazando Y por J -> JavadesdeCero
17 String Original: Ejemplo con espacios en blanco y tabs
18 Eliminando todos los espacios en blanco -> Ejemploconespacioenblancotabs

```

### 13.5. Arrays de String

Al igual que cualquier otro tipo de datos, los String se pueden ensamblar en arrays. Por ejemplo:

```

1 // Demostrando arrays de String
2 class StringArray
3 {
4 public static void main (String[] args)
5 {
6 String str[]={ "Java", "desde", "Cero" };
7
8 System.out.println("Array Original: ");
9 for (String s : str)
10 System.out.print(s+ " ");
11 System.out.println("\n");
12
13 //Cambiando un String
14 str[1] = "Curso";
15 str[2] = "Online";
16
17 System.out.println("Array Modificado: ");
18 for (String s : str)
19 System.out.print(s+ " ");
20 System.out.println("\n");
21 }
22 }

```

Se muestra el resultado de este programa:

```

1 Array Original:
2 JavadesdeCero
3
4 Array Modificado:
5 JavaCursoOnline

```

### 13.6. Los String son inmutables

El contenido de un objeto String es inmutable. Es decir, una vez creada, la secuencia de caracteres que compone la cadena **no se puede modificar**. Esta restricción permite a Java implementar cadenas de manera más eficiente. Aunque esto probablemente suene como un serio inconveniente, no lo es.

Cuando necesite una cadena que sea una variación de una que ya existe, simplemente cree una nueva cadena que contenga los cambios deseados. Como los objetos String no utilizados se recolectan de forma automática, ni siquiera tiene que preocuparse por lo que sucede con las cadenas descartadas. Sin embargo, debe quedar claro que las variables de referencia de cadena pueden, por supuesto, cambiar el objeto al que hacen referencia. Es solo que el contenido de un objeto string específico no se puede cambiar después de haber sido creado.

Para comprender completamente por qué las cadenas inmutables no son un obstáculo, utilizaremos otro de los métodos de String: `substring()`. El método `substring()` devuelve una nueva cadena que contiene una parte especificada de la cadena invocadora. Como se fabrica un nuevo objeto String que contiene la subcadena, la cadena original no se altera y la regla de inmutabilidad permanece intacta. La forma de `substring( )` que vamos a utilizar se muestra aquí:

```
1 String substring(int beginIndex, int endIndex)
```

Aquí, `beginIndex` especifica el índice inicial, y `endIndex` especifica el punto de detención. Aquí hay un programa que demuestra el uso de `substring( )` y el principio de cadenas inmutables:

```

1 // uso de substring()
2 class SubString {
3 public static void main (String[] args){
4 String str="Java desde Cero";
5
6 //Construyendo un substring
7 String substr=str.substring(5,15);
8
9 System.out.println("str: "+str);
10 System.out.println("substr: "+substr);
11 }
12 }
```

Salida:

```

1 str: Java desde Cero
2 substr: desde Cero
```

Como puede ver, la cadena original `str` no se modifica, y `substr` contiene la subcadena.

### 13.7. String en Argumentos de Línea de Comandos

Ahora que conoce la clase `String`, puede comprender el parámetro `args` en `main()` que ha estado en cada programa mostrado hasta ahora. Muchos programas aceptan lo que se llaman **argumentos de línea de comando**. Un argumento de línea de comandos es la información que sigue directamente el nombre del programa en la línea de comando cuando se ejecuta.

Para acceder a los argumentos de la línea de comandos dentro de un programa Java es bastante fácil: se almacenan como cadenas en la matriz `String` pasada a `main()`. Por ejemplo, el siguiente programa muestra todos los argumentos de línea de comandos con los que se llama:

```

1 // Mostrando Información de Línea de Comando
2 class DemoLC{
3 public static void main (String[] args){
4 System.out.println("Aquí se muestran "+ args.length
5 + " argumentos de línea de comando.");
6 System.out.println("Estos son: ");
7 for (int i=0; i<args.length; i++){
8 System.out.println("arg["+i+"]: "+args);
9 }
10 }
11 }
```

Si `DemoLC` se ejecuta de esta manera,

```
1 java DemoLC uno dos tres
```

verá la siguiente salida:

```

1 Aquí se muestran 3 argumentos de línea de comando.
2 Estos son:
3 arg[0]: uno
4 arg[1]: dos
5 arg[2]: tres
```

Observe que el primer argumento se almacena en el índice 0, el segundo argumento se almacena en el índice 1, y así sucesivamente.

Para tener una idea de la forma en que se pueden usar los argumentos de la línea de comandos, considere el siguiente programa. Se necesita un argumento de línea de comandos que especifique el nombre de una persona. Luego busca a través de una matriz bidimensional de cadenas para ese nombre. Si encuentra una coincidencia, muestra el número de teléfono de esa persona.

```

1 class Telefono {
2 public static void main (String[] args){
3 String numeros[][]={{ "Alex", "123-456"},
4 { "Juan", "145-478"},
5 { "Javier", "789-457"},
6 { "Maria", "784-554"}
7 };
8 int i;
9 if (args.length != 1){
10 System.out.println("Ejecute asi: java Telefono <nombre>");
11 } else {
12 for (i = 0; i < numeros.length; i++) {
13 System.out.println(numeros[i][0] + ": " + numeros);
14 break;
15 }
16 if (i == numeros.length){
17 System.out.println("Nombre no encontrado.");
18 }
19 }
20 }
21 }

```

Aquí hay una muestra de ejecución:

```

1 java Telefono Alex
2 Alex: 123-456

```

## 13.8. Concatenar cadenas en Java

### 13.8.1. Operador +

```

1 String s1 = "Hola,";
2 String s2 = "cómo estas?";
3 String s3 = s1 + s2;
4 System.out.println("String 1: " + s1);
5 System.out.println("String 2: " + s2);
6 System.out.println("Cadena resultante: " + s3);

```

salida:

```

1 String 1: Hola,
2 String 2: cómo estas?
3 Cadena resultante: Hola,cómo estas?

```

### 13.8.2. Método concat()

```

1 String s1 = "Hola,";
2 String s2 = "cómo estas?";
3 String s3 = s1.concat(s2);
4 System.out.println("String 1: " + s1);
5 System.out.println("String 2: " + s2);
6 System.out.println("Cadena resultante: " + s3);

```

salida:

```

1 String 1: Hola,
2 String 2: cómo estas?
3 Cadena resultante: Hola,cómo estas?

```

### 13.8.3. Método append de StringBuilder

```

1 String s3 = (new StringBuilder()).append("Hola,").append("cómo estas?").toString();
2 System.out.println("Cadena resultante: " + s3);

```

salida:

```

1 Cadena resultante: Hola,cómo estas?

```

### 13.8.4. ¿Cuándo usar cada uno?

Si usas la concatenación de Strings en un bucle, por ejemplo algo similar a esto:

```
1 String s = "";
2 for (int i = 0; i < 100; i++) {
3 s += ", " + i;
4 }
5 System.out.println(s);
```

Para el caso anterior, lo mejor sería utilizar el método `append` de `StringBuilder` en lugar del operador `+` porque es mucho más rápido y consume menos memoria.

Versión con `append`:

```
1 StringBuilder s = new StringBuilder();
2 for (int i = 1; i < 100; i++) {
3 s.append(", ").append(i);
4 }
5 System.out.println(s.toString());
```

Si solo tienes una sentencia similar a esta:

```
1 String s = "1, " + "2, " + "3, " + "4, " ...;
```

Entonces puedes usar el operador `+` sin problemas, porque el compilador usará `StringBuilder` automáticamente.

 24 de noviembre de 2025

## 14. 5.3 Ejercicios de la UD04

### 14.1. Arrays. Ejercicios de recorrido

1. (Estaturas) Escribir un programa que lea de teclado la estatura de 10 personas y las almacene en un array. Al finalizar la introducción de datos, se mostrarán al usuario los datos introducidos con el siguiente formato:

```
1 Persona 1: 1.85 m.
2 Persona 2: 1.53 m.
3 ...
4 Persona 10: 1.23 m.
```

2. (Lluvias) Se dispone de un fichero, de nombre *LluviasEnero.txt*, que contiene 31 datos correspondientes a las lluvias caídas en el mes de enero del pasado año. Se desea analizar los datos del fichero para averiguar:

- La lluvia total caída en el mes.
- La cantidad media de lluvias del mes.
- La cantidad más grande de lluvia caída en un solo día.
- Cual fue el día que más llovió.
- La cantidad más pequeña de lluvia caída en un solo día.
- Cual fue el día que menos llovió.
- En cuantos días no llovió nada.
- En cuantos días la lluvia superó la media.
- Si en la primera quincena del mes llovió más o menos que en la segunda.
- En cuantos días la lluvia fue menor que la del día siguiente.

Para resolver el problema se desarrollarán los siguientes métodos:

- a. `public static void leerArray (double v[], String nombreFichero)`, que rellena el array v con datos que se encuentran en el fichero especificado. El número de datos a leer vendrá determinado por el tamaño del array y no por la cantidad de datos que hay en el fichero.
- b. `public static double suma(double[] v)`, que devuelve la suma de los elementos del array v
- c. `public static double media(double v[])`, que devuelve la media de los elementos del array v. Se puede hacer uso del método del apartado anterior.
- d. `public static int contarMayorQueMedia(double v[])`, que devuelve la cantidad de elementos del array v que son mayores que la media. Se puede hacer uso del método del apartado anterior.
- e. `public static double maximo(double v[])`, que devuelve el valor más grande almacenado en el array v.
- f. `public static double minimo(double v[])`, que devuelve el valor más pequeño almacenado en el array v.
- g. `public static int posMaximo(double v[])`, que devuelve la posición del elemento más grande de v. Si éste se repite en el array es suficiente devolver la posición en que aparece por primera vez.
- h. `public static int posMinimo(double v[])`, que devuelve la posición del elemento más pequeño de v. Si éste se repite en el array es suficiente devolver la posición en que aparece por primera vez.
- i. `public static int contarApariciones(double v[], double x)`, que devuelve el número de veces que el valor x aparece en el array v.
- j. `public static double sumaParcial(double v[], int izq, int der)`, que devuelve la suma de los elementos del array v que están entre las posiciones *izq* y *der*.
- k. `public static int menoresQueElSiguiente(double v[])`, que devuelve el número de elementos de v que son menores que el elemento que tienen a continuación.

Además dispones de un archivo *Lluvias.java* (incompleto), que el alumnado deberá completar. El resultado debería ser similar a este:

```
1 La suma de las lluvias es 93,30 litros
2 La media de las lluvias es 3,01 litros
3 La máxima de las lluvias es 12,40 litros
4 La máxima de las lluvias fué el dia 17
5 La mínima de las lluvias es 0,00 litros
6 La mínima de las lluvias fué el dia 1
7 Ha habido un total de 16 dias sin lluvia
8 Ha habido un total de 11 dias en los que la lluvia ha superado la media
9 La segunda quincena ha llovido más que la otra
10 Ha habido 8 dias en los que ha llovido menos que el dia siguiente
```

3. (Dados) El lanzamiento de un dado es un experimento aleatorio en el que cada número tiene las mismas probabilidades de salir. Según esto, cuantas más veces lancemos el dado, más se igualarán las veces que aparece cada uno de los 6 números. Vamos a hacer un programa para comprobarlo.

- Generaremos un número aleatorio entre 1 y 6 un número determinado de veces (por ejemplo 100.000). Para ello puedes usar el método `random` de la clase `Math`.
- Tras cada lanzamiento incrementaremos un contador correspondiente a la cifra que ha salido. Para ello crearemos un array `veces` de 7 componentes, en el que el `veces[1]` servirá para contar las veces que sale un 1, `veces[2]` para contar las veces que sale un 2, etc. `veces[0]` no se usará.
- Cada, por ejemplo, 1.000 lanzamientos mostraremos por pantalla las estadísticas que indican que porcentaje de veces ha aparecido cada número en los lanzamientos hechos hasta ese momento. Por ejemplo:

```

1 Número de lanzamientos: 1000
2 1: 18 %
3 2: 14 %
4 3: 21 %
5 4: 10 %
6 5: 18 %
7 6: 19 %
8
9 Número de lanzamientos: 2000
10 ...

```

- Para el número de lanzamientos (100.000 en el ejemplo) y para la frecuencia con que se muestran las estadísticas (1.000 en el ejemplo) utilizaremos dos **constantes** enteras, de nombre `LANZAMIENTOS` y `FRECUENCIA`, de esta forma podremos variar de forma cómoda el modo en que probamos el programa.

4. (Invertir) Diseñar un método `public static int[] invertirArray(int[] v)`, que dado un array `v` devuelva otro con los elementos en orden inverso. Es decir, el último en primera posición, el penúltimo en segunda, etc.

Desde el método `main` crearemos e inicializaremos un array, llamaremos a `invertirArray` y mostraremos el array invertido.

Implementa un método que imprima por pantalla un Array `public static void imprimirArray(int[] v)`, y así poder imprimir el Array `v` (con espacios entre cada elemento) aquí tienes el resultado de imprimir el array con los 10 primeros números:

```
1 1 2 3 4 5 6 7 8 9 10
```

5. (SumasParciales) Se quiere diseñar un método `public static int[] sumaParcial(int[] v)`, que dado un array de enteros `v`, devuelva otro array de enteros `t` de forma que `t[i] = v[0] + v[1] + ... + v[i]`. Es decir:

```

1 t[0] = v[0]
2 t[1] = v[0] + v[1]
3 t[2] = v[0] + v[1] + v[2]
4 ...
5 t[10] = v[0] + v[1] + v[2] + ... + v[10]

```

Desde el método `main` crearemos e inicializaremos un array, llamaremos a `sumaParcial` y mostraremos el array resultante.

Ejemplo de salida, suponiendo que `v = {2, 4, 1, 0, 6}`:

```
1 El valor del array con sumas parciales es:
2 2 6 7 7 13
```

6. (Rotaciones) Rotar una posición a la derecha los elementos de un array consiste en mover cada elemento del array una posición a la derecha. El último elemento pasa a la posición 0 del array. Por ejemplo si rotamos a la derecha el array `{1, 2, 3, 4}` obtendríamos `{4, 1, 2, 3}`.

- Diseñar un método `public static void rotarDerecha(int[] v)`, que dado un array de enteros rote sus elementos un posición a la derecha.
- En el método `main` crearemos e inicializaremos un array y rotaremos sus elementos tantas veces como elementos tenga el array (mostrando cada vez su contenido), de forma que al final el array quedará en su estado original. Por ejemplo, si inicialmente el array contiene `{7, 3, 4, 2}`, el programa mostrará

```

1 Rotación 1: 2 7 3 4
2 Rotación 2: 4 2 7 3
3 Rotación 3: 3 4 2 7
4 Rotación 4: 7 3 4 2

```

- Diseña también un método para rotar a la izquierda y pruébalo de la misma forma.

7. (DosArrays) Desarrolla los siguientes métodos en los que intervienen dos arrays y pruébalos desde el método `main`

- `public static double[] sumaArraysIguals (double[] a, double[] b)` que dados dos arrays de `double` `a` y `b`, del mismo tamaño devuelva un array con la suma de los elementos de `a` y `b`, es decir, devolverá el array `{a[0]+b[0], a[1]+b[1], ...}`
- `public static double[] sumaArrays(double[] a, double[] b)`. Repite el ejercicio anterior pero teniendo en cuenta que `a` y `b` podrían tener longitudes distintas. En tal caso el número de elementos del array resultante coincidirá con la longitud del array de mayor tamaño.

## 14.2. Arrays. Ejercicios de búsqueda

1. (Lluvias – continuación). Queremos incorporar al programa la siguiente información:

- Cual fue el **primer** día del mes en que llovió exactamente 19 litros (si no hubo ninguno mostrar un mensaje por pantalla indicándolo)
- Cual fue el **último** día del mes en que llovió exáctamente 8 litros (si no hubo ninguno mostrar un mensaje por pantalla indicándolo)

Para ello desarrollarán los siguientes métodos:

- `public static int posPrimero(double[] v, double x)`, que devuelve la posición de la primera aparición de `x` en el array `v`. Si `x` no está en `v` el método devolverá -1. El método realizará una búsqueda ascendente para proporcionar el resultado.
- `public static int posUltimo(double[] v, double x)`, que devuelve la posición de la última aparición de `x` en el array `v`. Si `x` no está en `v` el método devolverá -1. El método realizará una búsqueda descendente para proporcionar el resultado.

2. (Tocayos) Disponemos de los nombres de dos grupos de personas (dos arrays de `String`). Dentro de cada grupo todas las personas tienen nombres distintos, pero queremos saber cuántas personas del primer grupo tienen algún tocayo en el segundo grupo, es decir, el mismo nombre que alguna persona del segundo grupo. Escribir un programa que resuelva el problema (inicializa los dos arrays con los valores que quieras y diseña los métodos que consideres necesarios).

Por ejemplo, si los nombres son {"miguel", "José", "ana", "maría"} y {"Ana", "luján", "juan", "José", "pepa", "ángela", "sofía", "andrés", "bartolo"}, el programa mostraría:

```
1 josé tiene tocayo en el segundo grupo.
2 ana tiene tocayo en el segundo grupo.
3 TOTAL: 2 personas del primer grupo tienen tocayo.
```

Optimiza el algoritmo para que no tenga en cuenta si se escribe el nombre en mayúsculas, minúsculas o cualquier combinación de mayúsculas y minúsculas.

3. (SumaDespuesImpar) Escribir un método (`sumaDespuesImpar`) que, dado un array de enteros, devuelva la suma de los elementos que aparecen tras el primer valor impar (la suma debe incluir el valor impar encontrado). Usar `main` para probar el método.

```
1 Para el array: {0,2,4,6,3,4,7,8,1}
2 El total a partir del primer impar es: 23
```

4. (HayPares) (En papel) Para determinar si existe algún valor par en un array se proponen varias soluciones. Indica cual/cuales son válidas para resolver el problema.

```
1 public static boolean hayPares1(int[] v) {
2 int i = 0;
3
4 while (i < v.length && v[i] % 2 != 0) {
5 i++;
6 }
7
8 if (v[i] % 2 == 0) {
9 return true;
10 } else {
11 return false;
12 }
13 }
14
15 }
```

```

1 public static boolean haypares2(int v[]) {
2
3 int i = 0;
4
5 while (i < v.length && v[i] % 2 != 0) {
6 i++;
7 }
8
9 if (i < v.length) {
10 return true;
11 } else {
12 return false;
13 }
14 }
15 }
```

```

1 public static boolean haypares3(int v[]) {
2
3 int i = 0;
4
5 while (v[i] % 2 != 0 && i < v.length) {
6 i++;
7 }
8
9 if (i < v.length) {
10 return true;
11 } else {
12 return false;
13 }
14 }
15 }
```

```

1 public static boolean haypares4(int v[]) {
2
3 int i = 0;
4
5 boolean encontrado = false;
6
7 while (i <= v.length && !encontrado) {
8
9 if (v[i] % 2 == 0) {
10 encontrado = true;
11 } else {
12 i++;
13 }
14 }
15
16 return encontrado;
17 }
18 }
```

```

1 public static boolean haypares5(int v[]) {
2
3 int i = 0;
4
5 boolean encontrado = false;
6
7 while (i < v.length && !encontrado) {
8
9 if (v[i] % 2 == 0) {
10 encontrado = true;
11 }
12 i++;
13 }
14
15 return encontrado;
16 }
17 }
```

```

1 public static boolean haypares6(int v[]) {
2
3 int i = 0;
4
5 while (i < v.length) {
6
7 if (v[i] % 2 == 0) {
8 return true;
9 } else {
10 return false;
11 }
12 }
13 }
14
15 }
```

```

1 public static boolean haypares7(int v[]) {
2
3 int i = 0;
4
5 while (i < v.length) {
6
7 if (v[i] % 2 == 0) {
8 return true;
9 }
10 i++;
11 }
12
13 return false;
14
15 }
16
17 }
```

5. (Capicúa) Escribir un método (`capicua`) para determinar si un array de palabras (`String`) es capicúa, esto es, si la primera y última palabra del array son la misma, la segunda y la penúltima palabras también lo son, y así sucesivamente. Escribir el método `main` para probar el método anterior.

```

1 El array: {"Ana", "Barcelona", "Casa", "David", "Elena", "David", "Casa", "Barcelona", "Ana"} es capicua?: true
2 El array: {"Amigo", "Besugo", "Cita", "Dia", "Enano", "Dia"} es capicua?: false
```

6. (Subsecuencia) Escribir un método (`subsecuencia`) que, dado un array, determine la posición de la primera subsecuencia del array que comprenda al menos tres números enteros consecutivos en posiciones consecutivas del array. De no existir dicha secuencia devolverá -1. Realiza al menos una prueba del método en el `main`.

Por ejemplo: en el array {23, 8, 12, 6, 7, **9, 10, 11**, 2} hay 3 números consecutivos en tres posiciones consecutivas, a partir de la posición 6 humana (5 del array): {9,10,11}

```

1 En el array: {23, 8, 12, 6, 7, 9, 10, 11, 1}
2 Posición de la subsecuencia: 6
```

7. (MismosValores) Se desea comprobar si dos arrays de `double` contienen los mismos valores, aunque sea en orden distinto. Para ello se ha escrito el siguiente método, que aparece incompleto:

```

1 public static boolean mismosValores(double[] v1, double[] v2) {
2
3 boolean encontrado = false;
4 int i = 0;
5
6 while (i < v1.length && !encontrado) {
7
8 boolean encontrado2 = false;
9 int j = 0;
10 while (j < v2.length && !encontrado2) {
11 if (v1[?] == v2[?]) {
12 encontrado2 = true;
13 i++;
14 } else {
15 ;
16 }
17 }
18 if (encontrado2 == ?) {
19 encontrado = true;
20 }
21 }
22
23 return !encontrado;
```

Completa el programa en los lugares donde aparece el símbolo **?**

### 14.3. Matrices

1. (Notas). Se dispone de una matriz (de doubles) que contiene las notas de una serie de alumnos en una serie de asignaturas. Cada fila corresponde a un alumno, mientras que cada columna corresponde a una asignatura. Desarrollar métodos para:

- Imprimir las notas alumno por alumno `imprimirXAlumno`.
- Imprimir las notas asignatura por asignatura `imprimirXAsignatura`.
- Imprimir la media de cada alumno `imprimirMediaXAlumno`.
- Imprimir la media de cada asignatura `imprimirMediaXAsignatura`.
- Indicar cual es la asignatura más fácil, es decir la de mayor nota media `indicarAsignaturaMasFacil`.
- ¿Hay algún alumno que suspenda todas las asignaturas? ¿Quién? `alumnosSuspendenTodas`
- ¿Hay alguna asignatura en la que suspendan todos los alumnos? ¿Cuál es? `asignaturasSuspendenTodos`

Generar la matriz (al menos 5x5) en el método main, rellenarla, y comprobar los métodos anteriores.

Ejemplo de ejecución:

```

1 {{5.3, 6.8, 8.2, 3.4, 5.0, 3.0},
2 {0.0, 4.5, 3.0, 2.8, 4.9, 2.0},
3 {7.8, 8.8, 9.8, 8.9, 9.9, 1.2},
4 {6.5, 5.5, 4.5, 7.8, 10.0, 2.2},
5 {5.5, 6.6, 5.6, 6.7, 7.0, 3.3}}
6
7 --- IMPRIMIR POR ALUMNOS ---
8 Notas para el Alumno 0: 5,30 6,80 8,20 3,40 5,00 3,00
9 Notas para el Alumno 1: 0,00 4,50 3,00 2,80 4,90 2,00
10 Notas para el Alumno 2: 7,80 8,80 9,80 8,90 9,90 1,20
11 Notas para el Alumno 3: 6,50 5,50 4,50 7,80 10,00 2,20
12 Notas para el Alumno 4: 5,50 6,60 5,60 6,70 7,00 3,30
13
14 --- IMPRIMIR POR ASIGNATURAS ---
15 Notas para la Asignatura 0: 5,30 0,00 7,80 6,50 5,50
16 Notas para la Asignatura 1: 6,80 4,50 8,80 5,50 6,60
17 Notas para la Asignatura 2: 8,20 3,00 9,80 4,50 5,60
18 Notas para la Asignatura 3: 3,40 2,80 8,90 7,80 6,70
19 Notas para la Asignatura 4: 5,00 4,90 9,90 10,00 7,00
20 Notas para la Asignatura 5: 3,00 2,00 1,20 2,20 3,30
21
22 --- IMPRIMIR MEDIA POR ALUMNO ---
23 Media para el Alumno 0: 5,28
24 Media para el Alumno 1: 2,87
25 Media para el Alumno 2: 7,73
26 Media para el Alumno 3: 6,08
27 Media para el Alumno 4: 5,78
28
29 --- IMPRIMIR MEDIA POR ASIGNATURA ---
30 Media para la Asignatura 0: 5,02
31 Media para la Asignatura 1: 6,44
32 Media para la Asignatura 2: 6,22
33 Media para la Asignatura 3: 5,92
34 Media para la Asignatura 4: 7,36
35 Media para la Asignatura 5: 2,34
36
37 --- INDICAR LA ASIGNATURA MÁS FACIL---
38 4
39
40 --- ¿HAY ALGUN ALUMNO QUE SUSPENDA TODAS? ---
41 El alumno 1 las suspende todas
42
43 --- ¿HAY ALGUNA ASIGNATURA QUE SUSPENDAN TODOS? ---
44 La asignatura 5 la suspenden todos

```

2. (Ventas). Una empresa comercializa 10 productos para lo cual tiene 5 distribuidores.

Los datos de ventas los tenemos almacenados en una matriz (de enteros) de 5 filas x 10 columnas, `ventas`, con el número de unidades de cada producto que ha vendido cada distribuidor. Cada fila corresponde a las ventas de un distribuidor (la primera fila, del primer distribuidor, etc.), mientras que cada columna corresponde a un producto :

| ventas  | columna[0] | columna[1] | columna[2] | columna[3] | columna[4] | columna[5] |
|---------|------------|------------|------------|------------|------------|------------|
| fila[0] | 100        | 25         | 33         | 89         | 23         | 90         |
| fila[1] | 28         | 765        | 65         | 77         | 987        | 55         |
| ...     | ...        |            |            |            |            |            |
| ...     |            |            |            |            |            |            |

El array `precio`, de 10 elementos, contiene el precio en € de cada uno de los 10 productos.

| precio[0] | precio[1] | precio[2] | ... |
|-----------|-----------|-----------|-----|
| 125.2     | 234.4     | 453.9     | ... |

Escribe el programa y los métodos necesarios para averiguar:

- Distribuidor que más artículos ha vendido `int distribuidorMasVende(int[][] matriz)` .
- El artículo que más se vende `int articuloMasVende(int[][] matriz)` .
- Sabiendo que los distribuidores que realizan ventas superiores a 30.000€ cobran una comisión del 5% de las ventas y los que superan los 70.000€ una comisión del 8%, emite un informe de los distribuidores que cobran comisión, indicando nº de distribuidor, importe de las ventas, porcentaje de comisión e importe de la comisión en € `void informe(int[][] ventas, double[] precio)` .

Ejemplo de ejecución

```

1 --- VENTAS DE DISTRIBUIDORES ---
2 Distribuidores (filas) y productos (columnas)
3 {
4 {5, 10, 15, 20, 25, 30, 35, 40, 45, 50},
5 {55, 45, 35, 25, 20, 10, 15, 25, 30, 50},
6 {25, 30, 35, 30, 25, 30, 35, 30, 25, 30},
7 {60, 59, 58, 57, 56, 55, 54, 53, 52, 51},
8 {10, 20, 30, 40, 50, 40, 30, 20, 10, 5}
9 }
10 -----
11 PRECIOS DE LOS PRODUCTOS
12 {55.0, 65.5, 85, 95.5, 55.5, 65, 85.5, 59, 510, 511}
13 -----
14 EL DISTRIBUIDOR QUE MÁS VENDE: 3
15 EL ARTÍCULO QUE MÁS SE VENDE: 9
16
17 --- INFORME DE VENTAS ---
18 El distribuidor número 1 ha vendido 61.305,00. Lo que supone un porcentaje de 5% que supone 3.065,25 €.
19 El distribuidor número 2 ha vendido 56.702,50. Lo que supone un porcentaje de 5% que supone 2.835,13 €.
20 El distribuidor número 3 ha vendido 45.360,00. Lo que supone un porcentaje de 5% que supone 2.268,00 €.
21 El distribuidor número 4 ha vendido 84.546,00. Lo que supone un porcentaje de 8% que supone 6.763,68 €.
22 El distribuidor número 5 ha vendido 25.005,00. Lo que no supone ninguna comisión.

```

3. (Utiles) Dada una matriz (de enteros) con el mismo número de filas y de columnas, diseñar los siguientes métodos:

- `public static void mostrarDiagonal(int[][] m)` que muestre por pantalla los elementos de la diagonal principal.
- `public static int filaDelMayor (int[][] m)`, que devuelva la fila en que se encuentra el mayor elemento de la matriz. (Si aparece más de una vez, el primero que aparezca de arriba a abajo)
- `public static void intercambiarFilas(int[][] m, int f1, int f2)`, que intercambie los elementos de las filas indicadas.
- Escribir un método `public static boolean esSimetrica (int[][] m)` que devuelva true si la matriz m es simétrica. Una matriz es simétrica si tiene el mismo número de filas que de columnas y además `m[i][j] = m[j][i]` para todo par de índices `i, j`.

### Acción

Todos los métodos se llaman y devuelven con los valores "reales" de las filas y columnas (0-based), pero los valores se muestra al usuario con los valores "amigables" (1-based)

### Nota

Puedes usar `Arrays.deepToString(m)` para imprimir las matrices

Por ejemplo, es simétrica:

```

1 1 5 3
2 5 4 7
3 3 7 5

```

Ejemplo de ejecución

```

1 MATRIZ:
2 {[{1, 5, 3},
3 {5, 4, 7},
4 {3, 7, 5}}
5 MOSTRAR DIAGONAL: 1 4 5
6 FILA DEL MAYOR: 2
7 ANTES DE INVERTIR LA FILA 1 Y LA 3
8 [[1, 5, 3], [5, 4, 7], [3, 7, 5]]
9 DESPUES DE INVERTIR LA FILA 1 Y LA 3
10 [[3, 7, 5], [5, 4, 7], [1, 5, 3]]
11 VOLVEMOS A INVERTIR LAS FILAS PARA CONSEGUIR LA MATRIZ ORIGINAL
12 [[1, 5, 3], [5, 4, 7], [3, 7, 5]]
13 LA MATRIZ ES SIMETRICA?: true

```

4. (Tetris) Supongamos que estamos desarrollando un Tetris en Java y para representar la partida utilizamos una matriz bidimensional de enteros 15 filas por 8 columnas. Se utiliza el valor 0 para indicar que la celda está vacía y un valor distinto de cero para las celdas que contienen parte de una pieza (distintos valores para distintos colores):



|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 2 | 2 | 2 | 2 | 2 | 0 |
| 1 | 0 | 2 | 4 | 4 | 0 | 3 | 0 |
| 1 | 1 | 2 | 4 | 4 | 3 | 3 | 3 |

Escribir un método que reciba la matriz y elimine las filas completas, haciendo caer las piezas que hay por encima de las celdas eliminadas tal y como se hace en el juego.

Métodos a implementar:

- `boolean esFilaLlena(int[] fila)`: devuelve true si la fila está llena (ningún cero)
- `void limpiarFilas(int[][] partida)`: Recorremos todas las filas de abajo hacia arriba, si la fila está llena: la eliminamos y bajamos todas las celdas superiores y volvemos a comprobar la fila donde estábamos. Sino seguimos hacia arriba.
- `void imprimePartida(int[][] partida)`: imprime la partida por consola

Ejemplo de ejecución1:

```

1 Comienzo1
2 [0, 0, 0, 0, 0, 0, 0]
3 [0, 0, 0, 0, 0, 0, 0]
4 [0, 0, 0, 0, 0, 0, 0]
5 [0, 0, 0, 0, 0, 0, 0]
6 [0, 0, 0, 0, 0, 0, 0]
7 [0, 0, 0, 0, 0, 0, 0]
8 [0, 0, 0, 0, 0, 0, 0]
9 [0, 0, 0, 0, 0, 0, 0]
10 [0, 0, 0, 0, 0, 0, 0]
11 [0, 0, 0, 0, 0, 0, 0]
12 [0, 0, 0, 0, 0, 0, 0]
13 [0, 0, 2, 0, 0, 0, 0]
14 [1, 0, 2, 2, 2, 2, 0]
15 [1, 0, 2, 4, 4, 0, 3]
16 [1, 1, 2, 4, 4, 3, 3]
17 Limpia1
18 [0, 0, 0, 0, 0, 0, 0]
19 [0, 0, 0, 0, 0, 0, 0]
20 [0, 0, 0, 0, 0, 0, 0]
21 [0, 0, 0, 0, 0, 0, 0]
22 [0, 0, 0, 0, 0, 0, 0]
23 [0, 0, 0, 0, 0, 0, 0]
24 [0, 0, 0, 0, 0, 0, 0]
25 [0, 0, 0, 0, 0, 0, 0]
26 [0, 0, 0, 0, 0, 0, 0]
27 [0, 0, 0, 0, 0, 0, 0]
28 [0, 0, 0, 0, 0, 0, 0]
29 [0, 0, 0, 0, 0, 0, 0]
30 [0, 0, 2, 0, 0, 0, 0]
31 [1, 0, 2, 2, 2, 2, 0]
32 [1, 0, 2, 4, 4, 0, 3]

```

### Ejemplo de ejecución 2:

```

1 Comienzo2
2 [0, 0, 0, 0, 0, 0, 0]
3 [0, 0, 0, 0, 0, 0, 0]
4 [0, 0, 0, 0, 0, 0, 0]
5 [0, 0, 0, 0, 0, 0, 0]
6 [0, 0, 0, 0, 0, 0, 0]
7 [0, 0, 0, 0, 0, 0, 0]
8 [0, 0, 0, 0, 0, 0, 0]
9 [0, 0, 0, 0, 0, 0, 0]
10 [0, 0, 0, 0, 0, 0, 0]
11 [0, 0, 0, 0, 0, 0, 0]
12 [0, 0, 0, 0, 0, 0, 0]
13 [0, 0, 2, 0, 0, 0, 0]
14 [1, 0, 2, 2, 2, 2, 0]
15 [1, 3, 2, 4, 4, 3, 4]
16 [1, 1, 2, 4, 4, 3, 3]
17 Limpia2
18 [0, 0, 0, 0, 0, 0, 0]
19 [0, 0, 0, 0, 0, 0, 0]
20 [0, 0, 0, 0, 0, 0, 0]
21 [0, 0, 0, 0, 0, 0, 0]
22 [0, 0, 0, 0, 0, 0, 0]
23 [0, 0, 0, 0, 0, 0, 0]
24 [0, 0, 0, 0, 0, 0, 0]
25 [0, 0, 0, 0, 0, 0, 0]
26 [0, 0, 0, 0, 0, 0, 0]
27 [0, 0, 0, 0, 0, 0, 0]
28 [0, 0, 0, 0, 0, 0, 0]
29 [0, 0, 0, 0, 0, 0, 0]
30 [0, 0, 0, 0, 0, 0, 0]
31 [0, 0, 2, 0, 0, 0, 0]
32 [1, 0, 2, 2, 2, 2, 0]

```

### Ejemplo de ejecución 3:

```

1 Comienzo3
2 [0, 0, 0, 0, 0, 0, 0]
3 [0, 0, 0, 0, 0, 0, 0]
4 [0, 0, 0, 0, 0, 0, 0]
5 [0, 0, 0, 0, 0, 0, 0]
6 [0, 0, 0, 0, 0, 0, 0]
7 [5, 5, 5, 5, 4, 4, 4]
8 [0, 0, 0, 4, 4, 0, 0]
9 [0, 0, 0, 4, 4, 0, 0]
10 [0, 0, 0, 4, 6, 0, 0]
11 [0, 0, 0, 4, 7, 0, 0]
12 [0, 0, 0, 4, 3, 0, 0]
13 [0, 0, 2, 0, 3, 2, 0, 0]
14 [1, 0, 2, 2, 2, 2, 0, 0]
15 [1, 3, 2, 4, 4, 3, 4]
16 [1, 1, 2, 4, 3, 3, 3]
17 Limpia3
18 [0, 0, 0, 0, 0, 0, 0]
19 [0, 0, 0, 0, 0, 0, 0]
20 [0, 0, 0, 0, 0, 0, 0]
21 [0, 0, 0, 0, 0, 0, 0]
22 [0, 0, 0, 0, 0, 0, 0]
23 [0, 0, 0, 0, 0, 0, 0]
24 [0, 0, 0, 0, 0, 0, 0]
25 [0, 0, 0, 0, 0, 0, 0]
26 [0, 0, 0, 4, 4, 0, 0]
27 [0, 0, 0, 4, 4, 0, 0]
28 [0, 0, 0, 4, 6, 0, 0]
29 [0, 0, 0, 4, 7, 0, 0]
30 [0, 0, 0, 4, 3, 0, 0]
31 [0, 0, 2, 0, 3, 2, 0, 0]
32 [1, 0, 2, 2, 2, 2, 0, 0]

```

## 14.4. Recursividad

1. (Palindromo) Implemente, tanto de forma recursiva (`boolean palindromoRecursivo(String frase)`) como de forma iterativa (`boolean palindromoIterativo(String frase)`), métodos que nos digan si una cadena de caracteres es simétrica (un palíndromo). Por ejemplo, "DABALEARROZALAZORRAELABAD" es un palíndromo.

"La ruta nos aporto otro paso natural"

"Nada, yo soy Adan"

"A mama Roma le aviva el amor a papa y a papa Roma le aviva el amor a mama"

"Ana, la tacaña catalana"

"Yo hago yoga hoy"

### Importante

Recuerda que no se debe tener en cuenta si la letra está en mayúsculas o minúsculas, solo si es la misma, por tanto `Ana` también es un palíndromo.

### Nota

¿Te atreves a implementar una solución que permita la entrada con espacios `PalindromoConEspacios`? ¿Y permitiendo espacios y signos de puntuación `PalindromoConEspaciosYPuntuacion`?"

2. (Voltear) Implemente, tanto de forma recursiva (`String recursiva(String frase)`) como de forma iterativa (`String iterativa(String frase)`), una función que le dé la vuelta a una cadena de caracteres.

### Importante

Obviamente, si la cadena es un palíndromo, la cadena y su inversa coincidirán.

Ejemplo de ejecución:

```

1 Introduce una frase para darle la vuelta: David es el mejor
2 Iterativa: rojem le se divad
3 Recursiva: rojem le se divad

```

3. (Combinaciones) Implemente, tanto de forma recursiva como de forma iterativa, una función que permitan calcular el número de combinaciones de  $n$  elementos tomados de  $m$  en  $m$ .

Realice dos versiones de la implementación iterativa, una aplicando la fórmula y otra utilizando una matriz auxiliar (en la que se vaya construyendo el triángulo de Pascal).

4. (MCD) Implemente, tanto de forma recursiva (`int recursiva(int m, int n)`) como de forma iterativa (`int iterativa(int m, int n)`), una función que nos devuelva el máximo común divisor de dos números enteros utilizando el algoritmo de Euclides.

```

1 ALGORITMO DE EUCLIDES MCD
2 Dados dos números enteros positivos m y n, tal que m > n, para encontrar su máximo común divisor (es decir, el mayor entero positivo que divide a
3 ambos):
4 - Dividir m por n para obtener el resto r (0 ≤ r < n)
5 - Si r = 0, el MCD es n.
 - Si no, el máximo común divisor es MCD(n,r).

```

5. (MergeSort) La ordenación por mezcla (mergesort) es un método de ordenación que se basa en un principio muy simple: se ordenan las dos mitades de un vector y, una vez ordenadas, se mezclan. Escriba un programa que implemente este método de ordenación.

6. (Descomposiciones) Diseñe e implemente un algoritmo que imprima todas las posibles descomposiciones de un número natural como suma de números menores que él (sumas con más de un sumando).

7. (Determinante) Diseñe e implemente un método recursivo que nos permita obtener el determinante de una matriz cuadrada de dimensión  $n$ .

8. (CifrasYLetras) Diseñe e implemente un programa que juegue al juego de cifras de "Cifras y Letras". El juego consiste en obtener, a partir de 6 números, un número lo más cercano posible a un número de tres cifras realizando operaciones aritméticas con los 6 números.

9. (OchoReinas) Problema de las 8 reinas: Se trata de buscar la forma de colocar 8 reinas en un tablero de ajedrez de forma que ninguna de ellas amenace ni se vea amenazada por otra reina.

```

1 Algoritmo:
2 - Colocar la reina i en la primera casilla válida de la fila i
3 - Si una reina no puede llegar a colocarse en ninguna casilla, se vuelve atrás y se cambia la posición de la reina de la fila i-1
4 - Intentar colocar las reinas restantes en las filas que quedan

```

10. (Laberinto) Salida de un laberinto: Se trata de encontrar un camino que nos permita salir de un laberinto definido en una matriz NxN. Para movernos por el laberinto, sólo podemos pasar de una casilla a otra que sea adyacente a la primera y no esté marcada como una casilla prohibida (esto es, las casillas prohibidas determinan las paredes que forman el laberinto).

```

1 Algoritmo:
2 - Se comienza en la casilla (0,0) y se termina en la casilla (N-1, N-1)
3 - Nos movemos a una celda adyacente si esto es posible.
4 - Cuando llegamos a una situación en la que no podemos realizar ningún movimiento que nos lleve a una celda que no hayamos visitado ya, retrocedemos sobre nuestros pasos y buscamos un camino alternativo.

```

## 15. 5.4 Talleres

---

### 15.1. Taller UD04\_01: GitHub Classroom

#### 15.1.1. Unirnos a GitHub Classroom

Aceptamos el *Assignement* (la tarea/ejercicio) a partir del link del profesor, en este caso: [https://classroom.github.com/a/bqGre\\_D](https://classroom.github.com/a/bqGre_D)

#### 15.1.2. Tarea

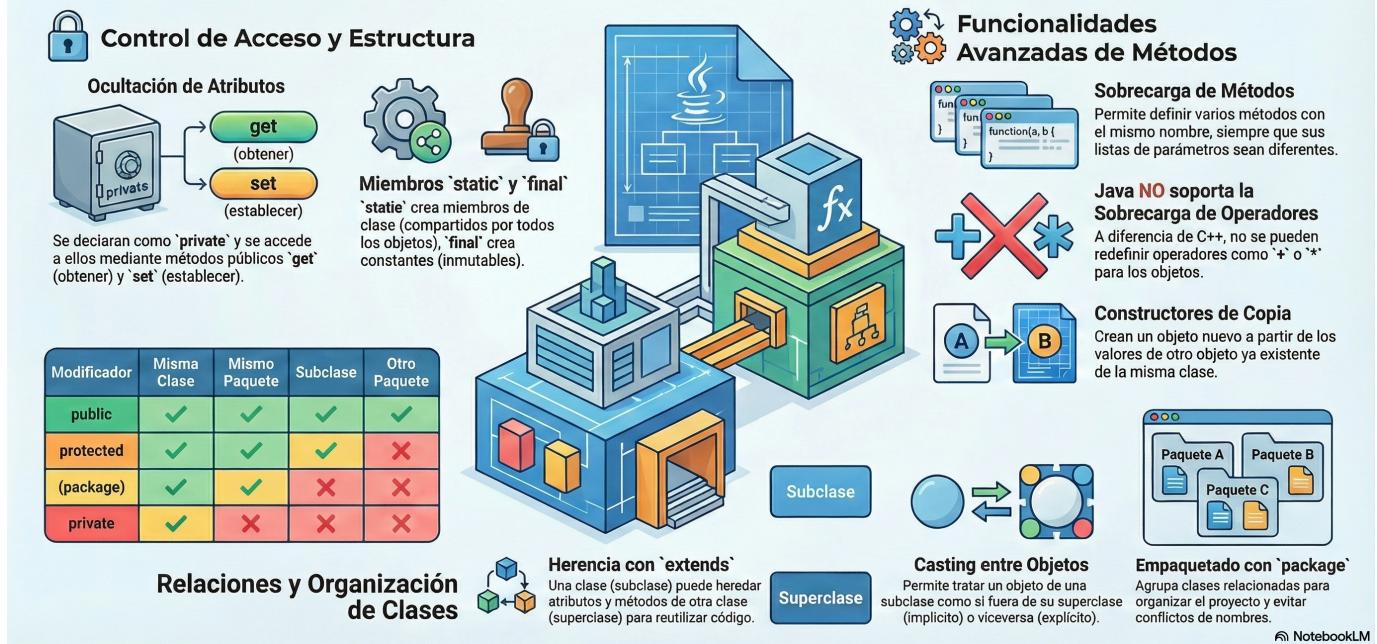
Debes enviar tus soluciones a GitHub Classroom y superar al menos la mitad de los tests, cuantos más tests superados, mejor nota tendrás en la tarea.

 30 de noviembre de 2025

## 6. UD05

### 16. 6.1 Desarrollo de clases

## Clases en Java: Conceptos Avanzados



### ⚠️ Cómo estudiar esta unidad?

Si lees esta unidad de principio a fin, verás que es como la [Unidad 2: Utilización de Objetos y Clases](#), pero con algunos conceptos más desarrollados y otros nuevos.

Si tienes absolutamente clara la Unidad 2, puedes leer solamente los siguientes puntos, que contienen las principales novedades. Si por el contrario tienes dudas, lagunas o algunos conceptos no quedaron claros, este es la última oportunidad de estudiarlos, preguntar al docente, entender los ejemplos y hacer los ejercicios. Desde esta unidad en adelante, los Objetos y Clases formaran parte del día a día, si te pierdes ahora será difícil seguir el ritmo.

Novedades respecto a la Unidad 2:

- 4.2. [Modificadores de acceso](#)
- 5.3. [Modificadores en la declaración de un método](#)
- 5.7. [Sobrecarga de operadores](#)
- 6.1 [Ocultación de atributos. Métodos de acceso](#)
- 6.2 [Ocultación de métodos](#)
- 8.4 [Constructores de copia](#)
- 9 [Clases Anidadas, Clases Internas \(Inner Class\)](#)
- 10 [Introducción a la herencia](#)
- 11 [Conversión entre objetos \(Casting\)](#)
- 12 [Acceso a métodos de la superclase](#)
- 13 [Empaquetado de clases](#)

## 16.1. Introducción

Como ya has visto en anteriores unidades, las clases están compuestas por atributos y métodos. Una clase especifica las características comunes de un conjunto de objetos.

De esta forma los programas que escribas estarán formados por un conjunto de clases a partir de las cuales irás creando objetos que se interrelacionarán unos con otros.

### Información

En esta unidad se va a utilizar el concepto de objeto así como algunas de las diversas estructuras de control básicas que ofrece cualquier lenguaje de programación. Todos esos conceptos han sido explicados y utilizados en las unidades anteriores. Si consideras que es necesario hacer un repaso del concepto de objeto o del uso de las estructuras de control elementales, éste es el momento de hacerlo.

### 16.1.1. Repaso del concepto de objeto

Desde el comienzo del módulo llevas utilizando el concepto de objeto para desarrollar tus programas de ejemplo. En las unidades anteriores se ha descrito un objeto como una entidad que contiene información y que es capaz de realizar ciertas operaciones con esa información. Según los valores que tenga esa información el objeto tendrá un estado determinado y según las operaciones que pueda llevar a cabo con esos datos serán responsables de un comportamiento concreto.

### Recuerda

Recuerda que entre las características fundamentales de un objeto se encontraban la **identidad** (los objetos son únicos y por tanto distinguibles entre sí, aunque pueda haber objetos exactamente iguales), un **estado** (los atributos que describen al objeto y los valores que tienen en cada momento) y un determinado **comportamiento** (acciones que se pueden realizar sobre el objeto).

Algunos ejemplos de objetos que podríamos imaginar podrían ser: - Un coche de color rojo, marca SEAT, modelo Toledo, del año 2003. En este ejemplo tenemos una serie de atributos, como el color (en este caso rojo), la marca, el modelo, el año, etc. Así mismo también podríamos imaginar determinadas características como la cantidad de combustible que le queda, o el número de kilómetros recorridos hasta el momento. - Un coche de color amarillo, marca Opel, modelo Astra, del año 2002. - Otro coche de color amarillo, marca Opel, modelo Astra y también del año 2002. Se trataría de otro objeto con las mismas propiedades que el anterior, pero sería un segundo objeto. - Un cocodrilo de cuatro metros de longitud y de veinte años de edad. - Un círculo de radio 2 centímetros, con centro en las coordenadas (0,0) y relleno de color amarillo. - Un círculo de radio 3 centímetros, con centro en las coordenadas (1,2) y relleno de color verde.

Si observas los ejemplos anteriores podrás distinguir sin demasiada dificultad al menos tres familias de objetos diferentes, que no tienen nada que ver una con otra:

- Los coches.
- Los círculos.
- Los cocodrilos.

Es de suponer entonces que cada objeto tendrá determinadas posibilidades de comportamiento (acciones) dependiendo de la familia a la que pertenezcan. Por ejemplo, en el caso de los coches podríamos imaginar acciones como: arrancar, frenar, acelerar, cambiar de marcha, etc. En el caso de los cocodrilos podrías imaginar otras acciones como: desplazarse, comer, dormir, cazar, etc. Para el caso del círculo se podrían plantear acciones como: cálculo de la superficie del círculo, cálculo de la longitud de la circunferencia que lo rodea, etc.

Por otro lado, también podrías imaginar algunos atributos cuyos valores podrían ir cambiando en función de las acciones que se realizaran sobre el objeto: ubicación del coche (coordenadas), velocidad instantánea, kilómetros recorridos, velocidad media, cantidad de combustible en el depósito, etc. En el caso de los cocodrilos podrías imaginar otros atributos como: peso actual, el número de dientes actuales (irá perdiendo algunos a lo largo de su vida), el número de presas que ha cazado hasta el momento, etc.

Como puedes ver, un objeto puede ser cualquier cosa que puedas describir en términos de atributos y acciones.

### Definición

Un objeto no es más que la representación de cualquier entidad concreta o abstracta que puedas percibir o imaginar y que pueda resultar de utilidad para modelar los elementos el entorno del problema que deseas resolver.

#### 16.1.2. El concepto de clase

Está claro que dentro de un mismo programa tendrás la oportunidad de encontrar decenas, cientos o incluso miles de objetos. En algunos casos no se parecerán en nada unos a otros, pero también podrás observar que habrá muchos que tengan un gran parecido, compartiendo un mismo comportamiento y unos mismos atributos. Habrá muchos objetos que sólo se diferenciaran por los valores que toman algunos de esos atributos.

Es aquí donde entra en escena el concepto de clase. Está claro que no podemos definir la estructura y el comportamiento de cada objeto cada vez que va a ser utilizado dentro de un programa, pues la escritura del código sería una tarea interminable y redundante. La idea es poder disponer de una plantilla o modelo para cada conjunto de objetos que sean del mismo tipo, es decir, que tengan los mismos atributos y un comportamiento similar.

### Definición

Una clase consiste en la definición de un tipo de objeto. Se trata de una descripción detallada de cómo van a ser los objetos que pertenezcan a esa clase indicando qué tipo de información contendrán (atributos) y cómo se podrá interactuar con ellos (comportamiento).

Como ya has visto en unidades anteriores, una clase consiste en un plantilla en la que se especifican:

- Los atributos que van a ser comunes a todos los objetos que pertenezcan a esa clase (información).
- Los métodos que permiten interactuar con esos objetos (comportamiento).

A partir de este momento podrás hablar ya sin confusión de objetos y de clases, sabiendo que los primeros son instancias concretas de las segundas, que no son más que una abstracción o definición.

Si nos volvemos a fijar en los ejemplos de objetos del apartado anterior podríamos observar que las clases serían lo que clasificamos como "familias" de objetos (coches, cocodrilos y círculos).

### Acción

En el lenguaje cotidiano de muchos programadores puede ser habitual la confusión entre los términos clase y objeto. Aunque normalmente el contexto nos permite distinguir si nos estamos refiriendo realmente a una clase (definición abstracta) o a un objeto (instancia concreta), hay que tener cuidado con su uso para no dar lugar a interpretaciones erróneas, especialmente durante el proceso de aprendizaje.

## 16.2. Estructura y miembros de una clase

En unidades anteriores ya se indicó que para declarar una clase en Java se usa la palabra reservada `class`. En la declaración de una clase vas a encontrar:

- **Cabecera de la clase.** Compuesta por una serie de modificadores de acceso, la palabra reservada `class` y el nombre de la clase.
- **Cuerpo de la clase.** En él se especifican los distintos miembros de la clase: atributos y métodos. Es decir, el contenido de la clase.

```

1 public class NombreDeLaClase [herencia] [interfaces]
2 {
3 // Atributos de la clase
4 ...
5 ...
6 ...
7 // Métodos de la clase
8 ...
9 ...
10 ...
11 }
```

Como puedes observar, el cuerpo de la clase es donde se declaran los atributos que caracterizan a los objetos de la clase y donde se define e implementa el comportamiento de dichos objetos; es decir, donde se declaran e implementan los métodos.

#### 16.2.1. Declaración de una clase.

La declaración de una clase en Java tiene la siguiente estructura general:

```

1 // Cabecera de la clase
2 [modificadores] class <NombreClase> [herencia] [interfaces] {
3 // Cuerpo de la clase
4 Declaración de los atributos
5 Declaración de los métodos
6 }
```

Un ejemplo básico pero completo podría ser:

```

1 class Punto{
2 // Atributos
3 private int x,y;
4
5 // Métodos
6 int obtenerX () {
7 return x;
8 }
9 int obtenerY() {
10 return y;
11 }
12 void establecerX (int nuevoX) {
13 x = nuevoX;
14 }
15 void establecerY (int nuevoY) {
16 y= nuevoY;
17 }
18 }
```

En este caso se trata de una clase muy sencilla en la que el cuerpo de la clase (el área entre las llaves) contiene el código y las declaraciones necesarias para que los objetos que se construyan (basándose en esta clase) puedan funcionar apropiadamente en un programa (declaraciones de atributos para contener el estado del objeto y métodos que implementen el comportamiento de la clase y los objetos creados a partir de ella).

Si te fijas en los distintos programas que se han desarrollado en los ejemplos de las unidades anteriores, podrás observar que cada uno de esos programas era en sí mismo una clase Java: se declaraban con la palabra reservada `class` y contenían algunos atributos (variables) así como algunos métodos (como mínimo el método `main`).

En el ejemplo anterior hemos visto lo mínimo que se tiene que indicar en la cabecera de una clase (el nombre de la clase y la palabra reservada `class`). Se puede proporcionar bastante más información mediante modificadores y otros indicadores como por ejemplo el nombre de su superclase (si es que esa clase hereda de otra), si implementa algún interfaz y algunas cosas más que irás aprendiendo poco a poco.

A la hora de implementar una clase Java (escribirla en un archivo con un editor de textos o con alguna herramienta integrada como por ejemplo Netbeans o Eclipse) debes tener en cuenta:

- Por convenio, se ha decidido que en lenguaje Java los nombres de las clases deben de empezar por una letra mayúscula. Así, cada vez que observes en el código una palabra con la primera letra en mayúscula sabrás que se trata de una clase sin necesidad de tener que buscar su declaración. Además, si el nombre de la clase está formado por varias palabras, cada una de ellas también tendrá su primera letra en mayúscula. Siguiendo esta recomendación, algunos ejemplos de nombres de clases podrían ser: `Recta`, `Circulo`, `Coche`, `CocheDeportivo`, `Jugador`, `JugadorFutbol`, `AnimalMarino`, `AnimalAcuatico`, etc.
- El archivo en el que se encuentra una clase Java debe tener el mismo nombre que esa clase si queremos poder utilizarla desde otras clases que se encuentren fuera de ese archivo (clase principal del archivo).
- Tanto la definición como la implementación de una clase se incluye en el mismo archivo (archivo `.java`). En otros lenguajes como por ejemplo C++, definición e implementación podrían ir en archivos separados (por ejemplo en C++, serían sendos archivos con extensiones `.h` y `.cpp`).

#### 16.2.2. Cabecera de una clase.

En general, la declaración de una clase puede incluir los siguientes elementos y en el siguiente orden:

1. Modificadores tales como `public`, `abstract` o `final`.
2. El nombre de la clase (con la primera letra de cada palabra en mayúsculas, por convenio).

3. El nombre de su clase padre (superclase), si es que se especifica, precedido por la palabra reservada `extends` ("extiende" o "hereda de").
4. Una lista separada por comas de interfaces que son implementadas por la clase, precedida por la palabra reservada `implements` ("implementa").
5. El cuerpo de la clase, encerrado entre llaves `{...}`.

La sintaxis completa de una cabecera (los cuatro primeros puntos) queda de la forma:

```
1 [modificadores] class <NombreClase> [extends <NombreSuperClase>][[implements <NombreInterface1>][[implements<NombreInterface2>] ...] {
```

En el ejemplo anterior de la clase `Punto` teníamos la siguiente cabecera:

```
1 class Punto {
```

En este caso no hay modificadores, ni indicadores de herencia, ni implementación de interfaces. Tan solo la palabra reservada `class` y el nombre de la clase. Es lo mínimo que puede haber en la cabecera de una clase.

La herencia y las interfaces las verás más adelante. Vamos a ver ahora cuáles son los modificadores que se pueden indicar al crear la clase y qué efectos tienen. Los modificadores de clase son:

```
1 [public] [final | abstract]
```

Veamos qué significado tiene cada uno de ellos:

- Modificador `public`. Indica que la clase es visible (se pueden crear objetos de esa clase) desde cualquier otra clase. Es decir, desde cualquier otra parte del programa. Si no se especifica este modificador, la clase sólo podrá ser utilizada desde clases que estén en el mismo paquete. El concepto de paquete lo veremos más adelante. **Sólo puede haber una clase public (clase principal) en un archivo .java**. El resto de clases que se definan en ese archivo no serán públicas.
- Modificador `abstract`. Indica que la clase es abstracta. **Una clase abstracta no es instanciable**. Es decir, no es posible crear objetos de esa clase (habrá que utilizar clases que hereden de ella). En este momento es posible que te parezca que no tenga sentido que esto pueda suceder (si no puedes crear objetos de esa clase, ¿para qué la quieres?), pero puede resultar útil a la hora de crear una jerarquía de clases. Esto lo verás también más adelante al estudiar el concepto de herencia.
- Modificador `final`. Indica que no podrás crear clases que hereden de ella. También volverás a este modificador cuando estudies el concepto de herencia. **Los modificadores final y abstract son excluyentes (sólo se puede utilizar uno de ellos)**.

Todos estos modificadores y palabras reservadas las iremos viendo poco a poco, así que no te preocupes demasiado por intentar entender todas ellas en este momento.

En el ejemplo anterior de la clase `Punto` tendríamos una clase que sería sólo visible (utilizable) desde el mismo paquete en el que se encuentra la clase (modificador de acceso por omisión o de paquete, o `package`). Desde fuera de ese paquete no sería visible o accesible. Para poder utilizarla desde cualquier parte del código del programa bastaría con añadir el atributo `public`:

```
1 public class Punto{
2 ...
3 }
```

### 16.2.3. Cuerpo de una clase.

Como ya has visto anteriormente, el cuerpo de una clase se encuentra encerrado entre llaves y contiene la declaración e implementación de sus miembros. Los miembros de una clase pueden ser:

- **Atributos**, que especifican los datos que podrá contener un objeto de la clase.
- **Métodos**, que implementan las acciones que se podrán realizar con un objeto de la clase.

Una clase puede no contener en su declaración atributos o métodos, pero debe de contener al menos uno de los dos (**la clase no puede ser vacía**).

En el ejemplo anterior donde se definía una clase `Punto`, tendríamos los siguientes atributos:

- Atributo `x`, de tipo `int`.
- Atributo `y`, de tipo `int`.

Es decir, dos valores de tipo entero. Cualquier objeto de la clase `Punto` que sea creado almacenará en su interior dos números enteros (`x` e `y`). Cada objeto diferente de la clase `Punto` contendrá sendos valores `x` e `y`, que podrán coincidir o no con el contenido de otros objetos de esa misma clase `Punto`.

Por ejemplo, si se han declarado varios objetos de tipo `Punto`:

```
1 Punto p1, p2, p3;
```

Sabremos que cada uno de esos objetos `p1`, `p2` y `p3` contendrán un par de coordenadas (`x`, `y`) que definen el estado de ese objeto. Puede que esos valores coincidan con los de otros objetos de tipo `Punto`, o puede que no, pero en cualquier caso serán objetos diferentes creados a partir del mismo molde (de la misma clase).

Por otro lado, la clase `Punto` también definía una serie de métodos:

- `java int obtenerX () { return x; }`
- `java int a;int obtenerY() { return y; }`
- `java void establecerX (int nuevoX) { x= nuevoX; }`
- `java void establecerY (int nuevoY) { y= nuevoY; }`

Cada uno de esos métodos puede ser llamado desde cualquier objeto que sea una instancia de la clase `Punto`. Se trata de operaciones que permiten manipular los datos (atributos) contenidos en el objeto bien para calcular otros datos o bien para modificar los propios atributos.

#### 16.2.4. Miembros estáticos o de clase.

Cada vez que se produce una instancia de una clase (es decir, se crea un objeto de esa clase), se desencadenan una serie de procesos (construcción del objeto) que dan lugar a la creación en memoria de un espacio físico que constituirá el objeto creado. De esta manera cada objeto tendrá sus propios miembros a imagen y semejanza de la plantilla propuesta por la clase.

Por otro lado, podrás encontrarte con ocasiones en las que determinados miembros de la clase (atributos o métodos) no tienen demasiado sentido como partes del objeto, sino más bien como partes de la clase en sí (partes de la plantilla, pero no de cada instancia de esa plantilla). Por ejemplo, si creamos una clase `coche` y quisieramos disponer de un atributo con el nombre de la clase (un atributo de tipo `String` con la cadena "Coche"), no tiene mucho sentido replicar ese atributo para todos los objetos de la clase `coche`, pues para todos va a tener siempre el mismo valor (la cadena "Coche"). Es más, ese atributo puede tener sentido y existencia al margen de la existencia de cualquier objeto de tipo `coche`. Podría no haberse creado ningún objeto de la clase `Coche` y sin embargo seguiría teniendo sentido poder acceder a ese atributo de nombre de la clase, pues se trata en efecto de un atributo de la propia clase más que de un atributo de cada objeto instancia de la clase.

Para poder definir miembros estáticos en Java se utiliza el modificador `static`. Los miembros (tanto atributos como métodos) declarados utilizando este modificador son conocidos como **miembros estáticos** o **miembros de clase**. A continuación vas a estudiar la creación y utilización de atributos y métodos. En cada caso verás cómo declarar y usar **atributos estáticos** y **métodos estáticos**.

### 16.3. Atributos

Los **atributos** constituyen la estructura interna de los objetos de una clase. Se trata del conjunto de datos que los objetos de una determinada clase almacenan cuando son creados. Es decir es como si fueran variables cuyo ámbito de existencia es el objeto dentro del cual han sido creadas. Fuera del objeto esas variables no tienen sentido y si el objeto deja de existir, esas variables también deberían hacerlo (proceso de destrucción del objeto). Los atributos a veces también son conocidos con el nombre de **variables miembro** o **variables de objeto**.

Los atributos pueden ser de cualquier tipo de los que pueda ser cualquier otra variable en un programa en Java: desde tipos elementales como `int`, `boolean` o `float` hasta tipos referenciados como `arrays`, `Strings` u `objetos`.

Además del tipo y del nombre, la declaración de un atributo puede contener también algunos modificadores (como por ejemplo `public`, `private`, `protected` o `static`). Por ejemplo, en el caso de la clase `Punto` que habíamos definido en el apartado anterior podrías haber declarado sus atributos como:

```
1 public int x;
2 public int y;
```

De esta manera estarías indicando que ambos atributos son públicos, es decir, accesibles por cualquier parte del código programa que tenga acceso a un objeto de esa clase.

## Acción

Como ya verás más adelante al estudiar el concepto de encapsulación, lo normal es declarar todos los atributos (o al menos la mayoría) como privados (`private`) de manera que si se desea acceder o manipular algún atributo se tenga que hacer a través de los métodos proporcionados por la clase.

### 16.3.1. Declaración de atributos.

La sintaxis general para la declaración de un atributo en el interior de una clase es:

```
1 [modificadores] <tipo> <nombreAtributo>;
```

Ejemplos:

```
1 int x;
2 public int elementoX, elementoY;
3 private int x1, y1, z1;
4 static double descuentoGeneral;
5 final boolean CASADO;
6 private Punto p1;
```

Te suena bastante, ¿verdad? La declaración de los atributos en una clase es exactamente igual a la declaración de cualquier variable tal y como has estudiado en las unidades anteriores y similar a como se hace en cualquier lenguaje de programación. Es decir mediante la indicación del tipo y a continuación el nombre del atributo, pudiéndose declarar varios atributos del mismo tipo mediante una lista de nombres de atributos separada por comas (exactamente como ya has estudiado al declarar variables).

La declaración de un atributo (o variable miembro o variable de objeto) consiste en la declaración de una variable que únicamente existe en el interior del objeto y por tanto su vida comenzará cuando el objeto comience a existir (el objeto sea creado). Esto significa que cada vez que se cree un objeto se crearán tantas variables como atributos contenga ese objeto en su interior (definidas en la clase, que es la plantilla o "molde" del objeto). Todas esas variables estarán encapsuladas dentro del objeto y sólo tendrán sentido dentro de él.

En el ejemplo que estamos utilizando de objetos de tipo `Punto` (instancias de la clase `Punto`), cada vez que se cree un nuevo `Punto` `p1`, se crearán sendos atributos `x`, `y` de tipo `int` que estarán en el interior de ese punto `p1`.

Si a continuación se crea un nuevo objeto `Punto` `p2`, se crearán otros dos nuevos atributos `x`, `y` de tipo `int` que estarán esta vez alojados en el interior de `p2`. Y así sucesivamente...

Dentro de la declaración de un atributo puedes encontrar tres partes:

- **Modificadores.** Son palabras reservadas que permiten modificar la utilización del atributo (indicar el control de acceso, si el atributo es constante, si se trata de un atributo de clase, etc.). Los iremos viendo uno a uno.
- **Tipo.** Indica el tipo del atributo. Puede tratarse de un tipo primitivo (`int`, `char`, `boolean`, `double`...) o bien de uno referenciado (`objeto`, `array`, etc.).
- **Nombre.** Identificador único para el nombre del atributo. Por convenio se suelen utilizar las minúsculas. En caso de que se trate de un identificador que contenga varias palabras, a partir de la segunda palabra se suele poner la letra de cada palabra en mayúsculas. Por ejemplo: `primerValor`, `valor`, `puertaIzquierda`, `cuartoTrasero`, `equipoVecendor`, `sumaTotal`, `nombreCandidatoFinal`, etc. Cualquier identificador válido de Java será admitido como nombre de atributo válido, pero es importante seguir este convenio para facilitar la legibilidad del código (todos los programadores de Java lo utilizan).

Como puedes observar, los atributos de una clase también pueden contener modificadores en su declaración (como sucedía al declarar la propia clase). Estos modificadores permiten indicar cierto comportamiento de una tributo a la hora de utilizarlo. Entre los modificadores de un atributo podemos distinguir:

- **Modificadores de acceso.** Indican la forma de acceso al atributo desde otra clase. Son modificadores excluyentes entre sí. Sólo se puede poner uno.
- **Modificadores de contenido.** No son excluyentes. Pueden aparecer varios a la vez.
- **Otros modificadores:** `transient` y `volatile`. El primero se utiliza para indicar que un atributo es transitorio (no persistente) y el segundo es para indicar al compilador que no debe realizar optimizaciones sobre esa variable. Es más que probable que no necesites utilizarlos en este módulo.

Aquí tienes la sintaxis completa de la declaración de un atributo teniendo en cuenta la lista de todos los modificadores e indicando cuáles son incompatibles unos con otros:

```
1 [private | protected | public] [static] [final] [transient] [volatile] <tipo><nombreAtributo>;
```

Vamos a estudiar con detalle cada uno de ellos.

### 16.3.2. Modificadores de acceso [NUEVO]

Los modificadores de acceso disponibles en Java para un atributo son:

- **Modificador de acceso public**. Indica que cualquier clase (por muy ajena o lejana que sea) tiene acceso a ese atributo. No es muy habitual declarar atributos públicos (`public`).
- **Modificador de acceso protected**. En este caso se permitirá acceder al atributo desde cualquier subclase (lo verás más adelante al estudiar la herencia) de la clase en la que se encuentre declarado el atributo, y también desde las clases del mismo paquete.
- **Modificador de acceso por omisión (o de paquete)**. Si no se indica ningún modificador de acceso en la declaración del atributo, se utilizará este tipo de acceso. Se permitirá el acceso a este atributo desde todas las clases que estén dentro del mismo paquete (`package`) que esta clase (la que contiene el atributo que se está declarando). No es necesario escribir ninguna palabra reservada. Si no se pone nada se supone se desea indicar este modo de acceso.
- **Modificador de acceso private**. Indica que sólo se puede acceder al atributo desde dentro de la propia clase. El atributo estará "oculto" para cualquier otra zona de código fuera de la clase en la que está declarado el atributo. Es lo opuesto a lo que permite `public`.

A continuación puedes observar un resumen de los distintos niveles accesibilidad que permite cada modificador:

| modificador                              | Misma clase | Mismo paquete | Subclase | Otro paquete |
|------------------------------------------|-------------|---------------|----------|--------------|
| public                                   | ✓           | ✓             | ✓        | ✓            |
| protected                                | ✓           | ✓             | ✓        | ✗            |
| Sin modificador ( <code>package</code> ) | ✓           | ✓             | ✗        | ✗            |
| private                                  | ✓           | ✗             | ✗        | ✗            |

#### Recuerda

¡Recuerda que **los modificadores de acceso son excluyentes!** Sólo se puede utilizar uno de ellos en la declaración de un atributo.

### 16.3.3. Modificadores de contenido.

Los modificadores de contenido no son excluyentes (pueden aparecer varios para un mismo atributo). Son los siguientes:

- **Modificador static**. Hace que el atributo sea común para todos los objetos de una misma clase. Es decir, todos los objetos de la clase compartirán ese mismo atributo con el mismo valor. Es un caso de **miembro estático** o **miembro de clase**: un **atributo estático** o **atributo de clase** o **variable de clase**.
- **Modificador final**. Indica que el atributo es una constante. Su valor no podrá ser modificado a lo largo de la vida del objeto. Por convenio, el nombre de los atributos constantes (`final`) se escribe con todas las letras en mayúsculas.

En el siguiente apartado sobre atributos estáticos verás un ejemplo completo de un atributo estático (`static`). Veamos ahora un ejemplo de atributo constante (`final`).

Imagina que estás diseñando un conjunto de clases para trabajar con expresiones geométricas (figuras, superficies, volúmenes, etc.) y necesitas utilizar muy a menudo la constante pi con abundantes cifras significativas, por ejemplo, 3.14159265. Utilizar esa constante literal muy a menudo puede resultar tedioso además de poco operativo (imagina que el futuro hubiera que cambiar la cantidad de cifras significativas). La idea es declararla una sola vez, asociarle un nombre simbólico (un identificador) y utilizar ese identificador cada vez que se necesite la constante. En tal caso puede resultar muy útil declarar un atributo final con el valor 3.14159265 dentro de la clase en la que se considere oportuno utilizarla. El mejor identificador que podrías utilizar para ella será probablemente el propio nombre de la constante (y en mayúsculas, para seguir el convenio de nombres), es decir, `PI`.

Así podría quedar la declaración del atributo:

```

1 class claseGeometria {
2 // Declaración de constantes
3 public final float PI = 3.14159265;
4 ...

```

#### 16.3.4. Atributos estáticos.

Como ya has visto, el modificador `static` hace que el atributo sea común (el mismo) para todos los objetos de una misma clase. En este caso sí podría decirse que la existencia del atributo no depende de la existencia del objeto, sino de la propia clase y por tanto sólo habrá uno, independientemente del número de objetos que se creen. El atributo será siempre el mismo para todos los objetos y tendrá un valor único independientemente de cada objeto. Es más, aunque no exista ningún objeto de esa clase, el atributo sí existirá y podrá contener un valor (pues se trata de un atributo de la clase más que del objeto).

Uno de los ejemplos más habituales (y sencillos) de atributos estáticos o de clase es el de un contador que indica el número de objetos de esa clase que se han ido creando. Por ejemplo, en la clase de ejemplo `Punto` podrías incluir un atributo que fuera ese contador para llevar un registro del número de objetos de la clase `Punto` que se van construyendo durante la ejecución del programa.

Otro ejemplo de atributo estático (y en este caso también constante) que también se ha mencionado anteriormente al hablar de miembros estáticos era disponer de un atributo `nombre`, que contuviera un `String` con el nombre de la clase. Nuevamente ese atributo sólo tiene sentido para la clase, pues habrá de ser compartido por todos los objetos que sean de esa clase (es el nombre de la clase a la que pertenecen los objetos y por tanto siempre será la misma e igual para todos, no tiene sentido que cada objeto de tipo `Punto` almacene en su interior el nombre de la clase, eso lo debería hacer la propia clase).

```

1 class Punto {
2 // Coordenadas del punto
3 private int x, y;
4 // Atributos de clase: cantidad de puntos creados hasta el momento
5 public static cantidadPuntos;

```

Obviamente, para que esto funcione como estás pensando, también habrá que escribir el código necesario para que cada vez que se cree un objeto de la clase `Punto` se incremente el valor del atributo `cantidadPuntos`.

#### Más información

Volverás a este ejemplo para implementar esa otra parte cuando estudies los constructores.

### 16.4. Métodos

Como ya has visto anteriormente, los métodos son las herramientas que nos sirven para definir el comportamiento de un objeto en sus interacciones con otros objetos. Forman parte de la estructura interna del objeto junto con los atributos.

En el proceso de declaración de una clase que estás estudiando ya has visto cómo escribir la cabecera de la clase y cómo especificar sus atributos dentro del cuerpo de la clase. Tan solo falta ya declarar los métodos, que estarán también en el interior del cuerpo de la clase junto con los atributos.

#### Importante

Los métodos suelen declararse después de los atributos. Aunque atributos y métodos pueden aparecer mezclados por todo el interior del cuerpo de la clase es aconsejable no hacerlo para mejorar la claridad y la legibilidad del código. De ese modo, cuando echemos un vistazo rápido al contenido de una clase, podremos ver rápidamente los atributos al principio (normalmente ocuparán menos líneas de código y serán fáciles de reconocer) y cada uno de los métodos inmediatamente después.

Cada método puede ocupar un número de líneas de código más o menos grande en función de la complejidad del proceso que pretenda implementar.

Los métodos representan la interfaz de una clase. Son la forma que tienen otros objetos de comunicarse con un objeto determinado solicitándole cierta información o pidiéndole que lleve a cabo una determinada acción. Este modo de programar, como ya has visto en unidades anteriores, facilita mucho la tarea al desarrollador de aplicaciones, pues le permite abstraerse del contenido de las clases haciendo uso únicamente del interfaz (métodos).

#### 16.4.1. Declaración de un método.

La definición de un método se compone de dos partes:

- **Cabecera** del método, que contiene el nombre del método junto con el tipo devuelto, un conjunto de posibles modificadores y una lista de parámetros.
- **Cuerpo** del método, que contiene las sentencias que implementan el comportamiento del método (incluidas posibles sentencias de declaración de variables locales).

Los elementos mínimos que deben aparecer en la declaración de un método son:

- El **tipo** devuelto por el método.
- El **nombre** del método.
- Los **paréntesis**.
- El **cuerpo** del método entre llaves: `{ }`.

Por ejemplo, en la clase `Punto` que se ha estado utilizando en los apartados anteriores podrías encontrar el siguiente método:

```
1 int obtenerX(){
2 // Cuerpo del método
3 ...
4 }
```

Donde:

- El **tipo** devuelto por el método es `int`.
- El **nombre** del método es `obtenerX`.
- **No recibe ningún parámetro**: aparece una lista vacía entre paréntesis: `( )`.
- El **cuerpo** del método es todo el código que habría encerrado entre llaves: `{ }`.

Dentro del cuerpo del método podrás encontrar declaraciones de variables, sentencias y todo tipo de estructuras de control (bucles, condiciones, etc.) que has estudiado en los apartados anteriores.

Ahora bien, la declaración de un método puede incluir algunos elementos más. Vamos a estudiar con detalle cada uno de ellos.

#### 16.4.2. Cabecera de método.

La declaración de un método puede incluir los siguientes elementos:

1. **Modificadores** (como por ejemplo los ya vistos `public` o `private`, más algunos otros que irás conociendo poco a poco). No es obligatorio incluir modificadores en la declaración.
2. El **tipo devuelto** (o tipo de retorno), que consiste en el tipo de dato (primitivo o referencia) que el método devuelve tras ser ejecutado. Si eliges `void` como tipo devuelto, el método no devolverá ningún valor.
3. El **nombre** del método, aplicándose para los nombres el mismo convenio que para los atributos.
4. Una **lista de parámetros** separados por comas y entre paréntesis donde cada parámetro debe ir precedido por su tipo. Si el método no tiene parámetros la lista estará vacía y únicamente aparecerán los paréntesis.
5. Una **lista de excepciones** que el método puede lanzar. Se utiliza la palabra reservada `throws` seguida de una lista de nombres de excepciones separadas por comas. No es obligatorio que un método incluya una lista de excepciones, aunque muchas veces será conveniente. En unidades anteriores ya has trabajado con el concepto de excepción y más adelante volverás a hacer uso de ellas.
6. El **cuerpo** del método, encerrado entre llaves. El cuerpo contendrá el código del método (una lista sentencias y estructuras de control en lenguaje Java) así como la posible declaración de variables locales.

La sintaxis general de la cabecera de un método podría entonces quedar así:

```
1 [private | protected | public] [static] [abstract] [final] [native] [synchronized] <tipo><nombreMétodo> ([<lista_parametros>]) [throws<lista_excepciones>]
```

Como sucede con todos los identificadores en Java (variables, clases, objetos, métodos, etc.), puede usarse cualquier identificador que cumpla las normas. Ahora bien, para mejorar la legibilidad del código, se ha establecido el siguiente convenio para nombrar los métodos: utilizar un verbo en minúscula o bien un nombre formado por varias palabras que comience por un verbo en minúscula, seguido por adjetivos, nombres, etc. los cuales sí aparecerán en mayúsculas.

Algunos ejemplos de métodos que siguen este convenio podrían ser: `ejecutar`, `romper`, `mover`, `subir`, `responder`, `obtenerX`, `establecerValor`, `estaVacio`, `estaLleno`, `moverFicha`, `subirPalanca`, `responderRapido`, `girarRuedaIzquierda`, `abrirPuertaDelantera`, `cambiarMarcha`, etc.

En el ejemplo de la clase `Punto`, puedes observar cómo los métodos `obtenerX` y `obtenerY` siguen el convenio de nombres para los métodos, devuelven en ambos casos un tipo `int`, su lista de parámetros es vacía (no tienen parámetros) y no lanzan ningún tipo de excepción:

- `java abstract int obtenerX()`
- `java int obtenerY()`

#### 16.4.3. Modificadores en la declaración de un método [NUEVO]

En la declaración de un método también pueden aparecer modificadores (como en la declaración de la clase o de los atributos). Un método puede tener los siguientes tipos de modificadores:

- **Modificadores de acceso.** Son los mismos que en el caso de los atributos (por omisión o de paquete (`package`), `public`, `private` y `protected`) y tienen el mismo cometido (acceso al método sólo por parte de clases del mismo paquete, o por cualquier parte del programa, o sólo para la propia clase, o también para las subclases).
- **Modificadores de contenido.** Son también los mismos que en el caso de los atributos (`static` y `final`) aunque su significado no es el mismo.
- **Otros modificadores** (no son aplicables a los atributos, sólo a los métodos): `abstract`, `native`, `synchronized`.

Un método `static` es un método cuya implementación es igual para todos los objetos de la clase y sólo tendrá acceso a los atributos estáticos de la clase (dado que se trata de un método de clase y no de objeto, sólo podrá acceder a la información de clase y no la de un objeto en particular). Este tipo de métodos pueden ser llamados sin necesidad de tener un objeto de la clase instanciado.

En Java un ejemplo típico de métodos estáticos se encuentra en la clase `Math`, cuyos métodos son todos estáticos (`Math.abs`, `Math.sin`, `Math.cos`, etc.). Como habrás podido comprobar en este ejemplo, la llamada a métodos estáticos se hace normalmente usando el nombre de la propia clase y no el de una instancia (objeto), pues se trata realmente de un método de clase. En cualquier caso, los objetos también admiten la invocación de los métodos estáticos de su clase y funcionaría correctamente.

Un método `final` es un método que no permite ser sobreescrito por las clases descendientes de la clase a la que pertenece el método. Volverás a ver este modificador cuando estudies en detalle la herencia.

El modificador `native` es utilizado para señalar que un método ha sido implementado en código nativo (en un lenguaje que ha sido compilado a lenguaje máquina, como por ejemplo C o C++). En estos casos simplemente se indica la cabecera del método, pues no tiene cuerpo escrito en Java.

Un método `abstract` (método abstracto) es un método que no tiene implementación (el cuerpo está vacío). La implementación será realizada en las clases descendientes. Un método sólo puede ser declarado como `abstract` si se encuentra dentro de una clase `abstract`. También volverás a este modificador en unidades posteriores cuando trabajes con la herencia.

Por último, si un método ha sido declarado como `synchronized`, el entorno de ejecución obligará a que cuando un proceso esté ejecutando ese método, el resto de procesos que tengan que llamar a ese mismo método deberán esperar a que el otro proceso termine. Puede resultar útil si sabes que un determinado método va a poder ser llamado concurrentemente por varios procesos a la vez. Por ahora no lo vas a necesitar.

Dada la cantidad de modificadores que has visto hasta el momento y su posible aplicación en la declaración de clases, atributos o métodos, veamos un resumen de todos los que has visto y en qué casos pueden aplicarse:

| modificador                              | Clase | Atributo | Método |
|------------------------------------------|-------|----------|--------|
| Sin modificador ( <code>package</code> ) | ✓     | ✓        | ✓      |
| <code>public</code>                      | ✓     | ✓        | ✓      |
| <code>private</code>                     | ✗     | ✓        | ✓      |
| <code>protected</code>                   | ✓     | ✓        | ✓      |
| <code>static</code>                      | ✗     | ✓        | ✓      |
| <code>final</code>                       | ✓     | ✓        | ✓      |
| <code>synchronized</code>                | ✗     | ✗        | ✓      |

| modificador | Clase | Atributo | Método |
|-------------|-------|----------|--------|
| native      | ✗     | ✗        | ✓      |
| abstract    | ✓     | ✗        | ✓      |

#### 16.4.4. Parámetros en un método.

La lista de parámetros de un método se coloca tras el nombre del método. Esta lista estará constituida por pares de la forma `<tipoParametro> <nombreParametro>`. Cada uno de esos pares estará separado por comas y la lista completa estará encerrada entre paréntesis:

```
1 <tipo> nombreMetodo (<tipo_1> <nombreParametro_1>, <tipo_2> <nombreParametro_2>, ..., <tipo_n><nombreParametro_n>)
```

Si la lista de parámetros es vacía, tan solo aparecerán los paréntesis:

```
1 <tipo> <nombreMetodo> ()
```

A la hora de declarar un método, debes tener en cuenta:

- Puedes incluir cualquier cantidad de parámetros. Se trata de una decisión del programador, pudiendo ser incluso una lista vacía.
- Los parámetros podrán ser de cualquier tipo (tipos primitivos, referencias, objetos, arrays, etc.).
- No está permitido que el nombre de una variable local del método coincida con el nombre de un parámetro.
- No puede haber dos parámetros con el mismo nombre. Se produciría ambigüedad.
- Si el nombre de algún parámetro coincide con el nombre de un atributo de la clase, éste será ocultado por el parámetro. Es decir, al indicar ese nombre en el código del método estarás haciendo referencia al parámetro y no al atributo. Para poder acceder al atributo tendrás que hacer uso del operador de autorreferencia `this`, que verás un poco más adelante.
- En Java el paso de parámetros es siempre por valor, excepto en el caso de los tipos referenciados (por ejemplo los objetos) en cuyo caso se está pasando efectivamente una referencia. La referencia (el objeto en sí mismo) no podrá ser cambiada pero sí elementos de su interior (atributos) a través de sus métodos o por acceso directo si se trata de un miembro público.

Es posible utilizar una construcción especial llamada `varargs` (argumentos variables) que permite que un método pueda tener un número variable de parámetros. Para utilizar este mecanismo se colocan unos puntos suspensivos (tres puntos: `...`) después del tipo del cual se puede tener una lista variable de argumentos, un espacio en blanco y a continuación el nombre del parámetro que aglutinará la lista de argumentos variables.

```
1 <tipo><nombreMetodo> (<tipo> ... <nombre>)
```

Es posible además mezclar el uso de `varargs` con parámetros fijos. En tal caso, la lista de parámetros variables debe aparecer al final (y sólo puede aparecer una). En realidad se trata una manera transparente de pasar un `array` con un número variable de elementos para no tener que hacerlo manualmente. Dentro del método habrá que ir recorriendo el `array` para ir obteniendo cada uno de los elementos de la lista de argumentos variables.

#### 16.4.5. Cuerpo de un método.

El interior de un método (cuerpo) está compuesto por una serie de sentencias en lenguaje Java:

- Sentencias de declaración de variables locales al método.
- Sentencias que implementan la lógica del método (estructuras de control como bucles o condiciones; utilización de métodos de otros objetos; cálculo de expresiones matemáticas, lógicas o de cadenas; creación de nuevos objetos, etc.). Es decir, todo lo que has visto en las unidades anteriores.
- Sentencia de devolución del valor de retorno (`return`). Aparecerá al final del método y es la que permite devolver la información que se le ha pedido al método. Es la última parte del proceso y la forma de comunicarse con la parte de código que llamó al método (paso de mensaje de vuelta). Esta sentencia de devolución siempre tiene que aparecer al final del método. Tan solo si el tipo devuelto por el método es `void` (vacío) no debe aparecer (pues no hay que devolver nada al código llamante).

En el ejemplo de la clase `Punto`, tenías los métodos `obtenerX` y `obtenerY`. Veamos uno de ellos:

```
1 int obtenerX(){
2 return x;
3 }
```

En ambos casos lo único que hace el método es precisamente devolver un valor (utilización de la sentencia `return`). No recibe parámetros (mensajes o información de entrada) ni hace cálculos, ni obtiene resultados intermedios o finales. Tan solo devuelve el contenido de un atributo. Se trata de uno de los métodos más sencillos que se pueden implementar: un método que devuelve el valor de un atributo. En inglés se les suele llamar métodos de tipo `get`, que en inglés significa `obtener`.

Además de esos dos métodos, la clase también disponía de otros dos que sirven para la función opuesta (`establecerX` y `establecerY`). Veamos uno de ellos:

```
1 void establecerX (int nuevoX){
2 x= nuevoX;
3 }
```

En este caso se trata de pasar un valor al método (parámetro `vx` de tipo `int`) el cual será utilizado para modificar el contenido del atributo `x` del objeto. Como habrás podido comprobar, ahora no se devuelve ningún valor (el tipo devuelto es `void` y no hay sentencia `return`). En inglés se suele hablar de métodos de tipo `set`, que en inglés significa poner o fijar (establecer un valor). El método `establecerY` es prácticamente igual pero para establecer el valor del atributo `y`.

Normalmente el código en el interior de un método será algo más complejo y estará formado un conjunto de sentencias en las que se realizarán cálculos, se tomarán decisiones, se repetirán acciones, etc. Puedes ver un ejemplo más completo en el siguiente ejercicio.

#### 16.4.6. Sobrecarga de métodos

En principio podrías pensar que un método puede aparecer una sola vez en la declaración de una clase (no se debería repetir el mismo nombre para varios métodos). Pero no tiene porqué siempre suceder así. Es posible tener varias versiones de un mismo método (varios métodos con el mismo nombre) gracias a la sobrecarga de métodos.

El lenguaje Java soporta la característica conocida como sobrecarga de métodos. Ésta permite declarar en una misma clase varias versiones del mismo método con el mismo nombre. La forma que tendrá el compilador de distinguir entre varios métodos que tengan el mismo nombre será mediante la lista de parámetros del método: si el método tiene una lista de parámetros diferente, será considerado como un método diferente (aunque tenga el mismo nombre) y el analizador léxico no producirá un error de compilación al encontrar dos nombres de método iguales en la misma clase.

Imagínate que estás desarrollando una clase para escribir sobre un lienzo que permite utilizar diferentes tipografías en función del tipo de información que se va a escribir. Es probable que necesitemos un método diferente según se vaya a pintar un número entero (`int`), un número real (`double`) o una cadena de caracteres (`String`). Una primera opción podría ser definir un nombre de método diferente dependiendo de lo que se vaya a escribir en el lienzo. Por ejemplo:

- Método `pintarEntero (int entero)`.
- Método `pintarReal (double real)`.
- Método `pintarCadena (double String)`.
- Método `pintarEnteroCadena (int entero, String cadena)`.

Y así sucesivamente para todos los casos que deseas contemplar...

La posibilidad que te ofrece la sobrecarga es utilizar un mismo nombre para todos esos métodos (dado que en el fondo hacen lo mismo: `pintar`). Pero para poder distinguir unos de otros será necesario que siempre exista alguna diferencia entre ellos en las listas de parámetros (bien en el número de parámetros, bien en el tipo de los parámetros). Volviendo al ejemplo anterior, podríamos utilizar un mismo nombre, por ejemplo `pintar`, para todos los métodos anteriores:

- Método `pintar (int entero)`.
- Método `pintar (double real)`.
- Método `pintar (double String)`.
- Método `pintar (int entero, String cadena)`.

En este caso el compilador no va a generar ningún error pues se cumplen las normas ya que unos métodos son perfectamente distinguibles de otros (a pesar de tener el mismo nombre) gracias a que tienen listas de parámetros diferentes.

Lo que sí habría producido un error de compilación habría sido por ejemplo incluir otro método `pintar (int entero)`, pues es imposible distinguirlo de otro método con el mismo nombre y con la misma lista de parámetros (ya existe un método `pintar` con un único parámetro de tipo `int`).

También debes tener en cuenta que el tipo devuelto por el método no es considerado a la hora de identificar un método, así que un tipo devuelto diferente no es suficiente para distinguir un método de otro. Es decir, no podrías definir dos métodos

exactamente iguales en nombre y lista de parámetros e intentar distinguirlos indicando un tipo devuelto diferente. El compilador producirá un error de duplicidad en el nombre del método y no te lo permitirá.

### Recuerda

Es conveniente no abusar de sobrecarga de métodos y utilizarla con cierta moderación (cuando realmente puede beneficiar su uso), dado que podría hacer el código menos legible.

#### 16.4.7. Sobre carga de operadores. [NUEVO]

Del mismo modo que hemos visto la posibilidad de sobre cargar métodos (disponer de varias versiones de un método con el mismo nombre cambiando su lista de parámetros), podría plantearse también la opción de sobre cargar operadores del lenguaje tales como `+`, `-`, `*`, `( )`, `<`, `>`, etc. para darles otro significado dependiendo del tipo de objetos con los que vaya a operar.

En algunos casos puede resultar útil para ayudar a mejorar la legibilidad del código, pues esos operadores resultan muy intuitivos y pueden dar una idea rápida de cuál es su funcionamiento.

Un típico ejemplo podría ser el de la sobre carga de operadores aritméticos como la suma (`+`) o el producto (`*`) para operar con fracciones. Si se definen objetos de una clase `Fracción` (que contendrá los atributos `numerador` y `denominador`) podrían sobre cargarse los operadores aritméticos (habría que redefinir el operador suma (`+`) para la suma, el operador asterisco (`*`) para el producto, etc.) para esta clase y así podrían utilizarse para sumar o multiplicar objetos de tipo `Fraccion` mediante el algoritmo específico de suma o de producto del objeto `Fraccion` (pues esos operadores no están preparados en el lenguaje para operar con esos objetos).

En algunos lenguajes de programación como por ejemplo C++ o C# se permite la sobre carga, pero no es algo soportado en todos los lenguajes. ¿Qué sucede en el caso concreto de Java?

El lenguaje Java **NO** soporta la sobre carga de operadores.

En el ejemplo anterior de los objetos de tipo Fracción, habrá que declarar métodos en la clase `Fraccion` que se encarguen de realizar esas operaciones, pero no lo podemos hacer sobre cargando los operadores del lenguaje (los símbolos de la suma, resta, producto, etc.). Por ejemplo:

```
1 public Fraccion sumar (Fraccion sumando)
2 public Fraccion multiplicar (Fraccion multiplicando)
```

Y así sucesivamente...

Dado que en este módulo se está utilizando el lenguaje Java para aprender a programar, no podemos hacer uso de esta funcionalidad. Más adelante, cuando aprendas a programar en otros lenguajes, es posible que sí tengas la posibilidad de utilizar este recurso.

#### 16.4.8. La referencia `this`.

La palabra reservada `this` consiste en una referencia al objeto actual. El uso de este operador puede resultar muy útil a la hora de evitar la ambigüedad que puede producirse entre el nombre de un parámetro de un método y el nombre de un atributo cuando ambos tienen el mismo identificador (mismo nombre). En tales casos el parámetro "oculta" al atributo y no tendríamos acceso directo a él (al escribir el identificador estaríamos haciendo referencia al parámetro y no al atributo). En estos casos la referencia `this` nos permite acceder a estos atributos ocultados por los parámetros.

Dado que `this` es una referencia a la propia clase en la que te encuentras en ese momento, puedes acceder a sus atributos mediante el operador punto (`.`) como sucede con cualquier otra clase u objeto. Por tanto, en lugar de poner el nombre del atributo (que estos casos haría referencia al parámetro), podrías escribir `this.nombreAtributo`, de manera que el compilador sabrá que te estás refiriendo al atributo y se eliminará la ambigüedad.

En el ejemplo de la clase `Punto`, podríamos utilizar la referencia `this` si el nombre del parámetro del método coincidiera con el del atributo que se desea modificar. Por ejemplo:

```

1 class Punto{
2 private int x,y;
3
4 void establecerX (int nuevaX){
5 int x = 1; //<<<---- metodo
6 this.x = 1; //<<<---- clase
7 this.x=x;
8 this.x=nuevaX;
9 }
10 }

```

En este caso ha sido indispensable el uso de `this`, pues si no sería imposible saber en qué casos te estás refiriendo al parámetro `x` y en cuáles al atributo `x`. Para el compilador el identificador `x` será siempre el parámetro, pues ha "ocultado" al atributo.

#### Más información

En algunos casos puede resultar útil hacer uso de la referencia `this` aunque no sea necesario, pues puede ayudar a mejorar la legibilidad del código.

#### 16.4.9. Métodos estáticos.

Como ya has visto en ocasiones anteriores, un método estático es un método que puede ser usado directamente desde la clase, sin necesidad de tener que crear una instancia para poder utilizar al método. También son conocidos como **métodos de clase** (como sucedía con los atributos de clase), frente a los métodos de objeto (es necesario un objeto para poder disponer de ellos).

Los métodos estáticos no pueden manipular atributos de instancias (objetos) sino atributos estáticos (de clase) y suelen ser utilizados para realizar operaciones comunes a todos los objetos de la clase, más que para una instancia concreta.

Algunos ejemplos de operaciones que suelen realizarse desde métodos estáticos:

- **Acceso a atributos específicos de clase:** incremento o decremento de contadores internos de la clase (`node_instancias`), acceso a un posible atributo de nombre de la clase, etc.
- **Operaciones genéricas relacionadas con la clase pero que no utilizan atributos de instancia.** Por ejemplo una clase `NIF` (o `DNI`) que permite trabajar con el `DNI` y la letra del `NIF` y que proporciona funciones adicionales para calcular la letra `NIF` de un número de `DNI` que se le pase como parámetro. Ese método puede ser interesante para ser usado desde fuera de la clase de manera independiente a la existencia de objetos de tipo `NIF`.

En la biblioteca de Java es muy habitual encontrarse con clases que proporcionan métodos estáticos que pueden resultar muy útiles para cálculos auxiliares, conversiones de tipos, etc. Por ejemplo, la mayoría de las clases del paquete `java.lang` que representan tipos (`Integer`, `String`, `Float`, `Double`, `Boolean`, etc.) ofrecen métodos estáticos para hacer conversiones. Aquí tienes algunos ejemplos:

- `java static String valueOf(int i)`  
Devuelve la representación en formato `String` (cadena) de un valor `int`. Se trata de un método que no tiene que ver nada en absoluto con instancias de concretas de `String`, sino de un método auxiliar que puede servir como herramienta para ser usada desde otras clases. Se utilizaría directamente con el nombre de la clase. Por ejemplo:
- `java String enteroCadena = String.valueOf(23).`
- `java static String valueOf(float f)`

Algo similar para un valor de tipo `float`. Ejemplo de uso:

- `java String floatCadena = String.valueOf(24.341)`
- `java static int parseInt(String s)`

En este caso se trata de un método estático de la clase `Integer`. Analiza la cadena pasada como parámetro y la transforma en un `int`. Ejemplo de uso:

- `java int cadenaEntero=Integer.parseInt ("-12")`

Todos los ejemplos anteriores son casos en los que se utiliza directamente la clase como una especie de caja de herramientas que contiene métodos que pueden ser utilizados desde cualquier parte, por eso suelen ser métodos públicos.

## 16.5. Encapsulación, control de acceso y visibilidad.

Dentro de la Programación Orientada a Objetos ya has visto que es muy importante el concepto de ocultación, la cual ha sido lograda gracias a la encapsulación de la información dentro de las clases. De esta manera una clase puede ocultar parte de su contenido o restringir el acceso a él para evitar que sea manipulado de manera inadecuada. Los modificadores de acceso en Java permiten especificar el ámbito de visibilidad de los miembros de una clase, proporcionando así un mecanismo de accesibilidad a varios niveles.

Acabas de estudiar que cuando se definen los miembros de una clase (atributos o métodos), e incluso la propia clase, se indica (aunque sea por omisión) un modificador de acceso. En función de la visibilidad que se desee que tengan los objetos o los miembros de esos objetos se elegirá alguno de los modificadores de acceso que has estudiado. Ahora que ya sabes cómo escribir una clase completa (declaración de la clase, declaración de sus atributos y declaración de sus métodos), vamos a hacer un repaso general de las opciones de visibilidad (control de acceso) que has estudiado.

Los modificadores de acceso determinan si una clase puede utilizar determinados miembros (acceder a atributos o invocar miembros) de otra clase. Existen dos niveles de control de acceso:

1. A nivel general (**nivel de clase**): visibilidad de la propia clase.
2. A **nivel de miembros**: especificación, miembro por miembro, de su nivel de visibilidad.

En el caso de la clase, ya estudiaste que los niveles de visibilidad podían ser:

- Público (modificador `public`), en cuyo caso la clase era visible a cualquier otra clase (cualquier otro fragmento de código del programa).
- Privada al paquete ( `package` )(sin modificador o modificador "por omisión"). En este caso, la clase sólo será visible a las demás clases del mismo paquete, pero no al resto del código del programa (otros paquetes).
- Protegido ( `protected` ), lo podrán ver las clases del mismo paquete y también las clases herederas.

En el caso de los miembros, disponías de otras dos posibilidades más de niveles de accesibilidad, teniendo un total de cuatro opciones a la hora de definir el control de acceso al miembro:

- Público (modificador `public`), igual que en el caso global de la clase y con el mismo significado (miembro visible desde cualquier parte del código).
- Del paquete (sin modificador), también con el mismo significado que en el caso de la clase (miembro visible sólo desde clases del mismo paquete, ni siquiera será visible desde una subclase salvo si ésta está en el mismo paquete).
- Privado (modificador `private`), donde sólo la propia clase tiene acceso al miembro.
- Protegido (modificador `protected`), lo podrán ver las clases del mismo paquete y también las clases herederas.

### 16.5.1. Ocultación de atributos. Métodos de acceso. [NUEVO]

Los atributos de una clase suelen ser declarados como privados a la clase o, como mucho, `protected` (accesibles también por clases heredadas), pero no como `public`. De esta manera puedes evitar que sean manipulados inadecuadamente (por ejemplo modificarlos sin ningún tipo de control) desde el exterior del objeto.

En estos casos lo que se suele hacer es declarar esos atributos como privados o protegidos y crear métodos públicos que permitan acceder a esos atributos. Si se trata de un atributo cuyo contenido puede ser observado pero no modificado directamente, puede implementarse un método de "obtención" del atributo (en inglés se les suele llamar método de tipo `get`) y si el atributo puede ser modificado, puedes también implementar otro método para la modificación o "establecimiento" del valor del atributo (en inglés se le suele llamar método de tipo `set`). Esto ya lo has visto en apartados anteriores.

Si recuerdas la clase `Punto` que hemos utilizado como ejemplo, ya hiciste algo así con los métodos de obtención y establecimiento de las coordenadas:

```

1 private int x, y;
2
3 // Métodos get
4 public int obtenerX() {
5 return x;
6 }
7 public int obtenerY() {
8 return y;
9 }
10 // Métodos set
11 public void establecerX(int x) {
12 this.x= x;
13 }
14 public void establecerY(int y) {
15 this.y= y;
16 }
```

Así, para poder obtener el valor del atributo `x` de un objeto de tipo `Punto` será necesario utilizar el `método obtenerX()` y no se podrá acceder directamente al atributo `x` del objeto. En algunos casos los programadores directamente utilizan nombres en inglés para nombrar a estos métodos:

```
1 getX(), getY(), setX(), setY(), getNombre, setNombre, getColor, etc.
```

También pueden darse casos en los que no interesa que pueda observarse directamente el valor de un atributo, sino un determinado procesamiento o cálculo que se haga con el atributo (pero no el valor original). Por ejemplo podrías tener un atributo `DNI` que almacene los 8 dígitos del `DNI` pero no la letra del `NIF` (pues se puede calcular a partir de los dígitos). El método de acceso para el `DNI` (método `getDNI`) podría proporcionar el `DNI` completo (es decir, el `NIF`, incluyendo la letra), mientras que la letra no es almacenada realmente en el atributo del objeto. Algo similar podría suceder con el dígito de control de una cuenta bancaria, que puede no ser almacenado en el objeto, pero sí calculado y devuelto cuando se nos pide el número de cuenta completo.

En otros casos puede interesar disponer de métodos de modificación de un atributo pero a través de un determinado procesamiento previo para por ejemplo poder controlar errores o valores inadecuados. Volviendo al ejemplo del `NIF`, un método para modificar un `DNI` (método `setDNI`) podría incluir la letra (`NIF` completo), de manera que así podría comprobarse si el número de `DNI` y la letra coinciden (es un `NIF` válido). En tal caso se almacenará el `DNI` y en caso contrario se producirá un error de validación (por ejemplo lanzando una excepción). En cualquier caso, el `DNI` que se almacenara sería solamente el número y no la letra (pues la letra es calculable a partir del número de `DNI`).

### 16.5.2. Ocultación de métodos. [NUEVO]

Normalmente los métodos de una clase pertenecen a su interfaz y por tanto parece lógico que sean declarados como públicos. Pero también es cierto que pueden darse casos en los que exista la necesidad de disponer de algunos métodos privados a la clase. Se trata de métodos que realizan operaciones intermedias o auxiliares y que son utilizados por los métodos que sí forman parte de la interfaz. Ese tipo de métodos (de comprobación, de adaptación de formatos, de cálculos intermedios, etc.) suelen declararse como privados pues no son de interés (o no es apropiado que sean visibles) fuera del contexto del interior del objeto.

En el ejemplo anterior de objetos que contienen un `DNI`, será necesario calcular la letra correspondiente a un determinado número de `DNI` o comprobar si una determinada combinación de número y letra forman un `DNI` válido. Este tipo de cálculos y comprobaciones podrían ser implementados en métodos privados de la clase (o al menos como métodos protegidos).

```
1 public static boolean validarDNI(String dni){
2 [...]
3 char letra = LetraDNI(Long numeroDNI);
4 [...]
5 }
6
7 ??? static char LetraDNI (Long numero) {
8 [...]
9 }
```

## 16.6. Utilización de los métodos y atributos de una clase.

Una vez que ya tienes implementada una clase con todos sus atributos y métodos, ha llegado el momento de utilizarla, es decir, de instanciar objetos de esa clase e interaccionar con ellos. En unidades anteriores ya has visto cómo declarar un objeto de una clase determinada, instanciarlo con el operador `new` y utilizar sus métodos y atributos.

### 16.6.1. Declaración de un objeto.

Como ya has visto en unidades anteriores, la declaración de un objeto se realiza exactamente igual que la declaración de una variable de cualquier tipo:

```
1 <tipo> nombreVariable;
```

En este caso el tipo será alguna clase que ya hayas implementado o bien alguna de las proporcionadas por la biblioteca de Java o por alguna otra biblioteca escrita por terceros.

Por ejemplo:

```
1 Punto p1;
2 Rectangulo r1, r2;
3 Coche cocheAntonio;
4 String palabra;
```

Esas variables (`p1`, `r1`, `r2`, `cocheAntonio`, `palabra`) en realidad son referencias (también conocidas como punteros o direcciones de memoria) que apuntan (hacen "referencia") a un objeto (una zona de memoria) de la clase indicada en la declaración.

Como ya estudiaste en la unidad dedicada a los objetos, un objeto recién declarado (referencia recién creada) no apunta a nada. Se dice que la referencia está vacía o que es una referencia nula (la variable `objeto` contiene el valor `null`). Es decir, la variable existe y está preparada para guardar una dirección de memoria que será la zona donde se encuentre el objeto al que hará referencia, pero el objeto aún no existe (no ha sido creado o instanciado). Por tanto se dice que apunta a un objeto nulo o inexistente.

Para que esa variable (referencia) apunte realmente a un objeto (contenga una referencia o dirección de memoria que apunte a una zona de memoria en la que se ha reservado espacio para un objeto) es necesario crear o instanciar el objeto. Para ello se utiliza el operador `new`.

### 16.6.2. Creación de un objeto.

Para poder crear un objeto (instancia de una clase) es necesario utilizar el operador `new`, el cual tiene la siguiente sintaxis:

```
1 nombreObjeto= new <ConstructorClase> ([listaParametros]);
```

El constructor de una clase (`<ConstructorClase>`) es un método especial que tiene toda clase y cuyo nombre coincide con el de la clase. Es quien se encarga de crear o construir el objeto, solicitando la reserva de memoria necesaria para los atributos e inicializándolos a algún valor si fuera necesario.

Dado que el constructor es un método más de la clase, podrá tener también su lista de parámetros como tienen todos los métodos.

De la tarea de reservar memoria para la estructura del objeto (sus atributos más alguna otra información de carácter interno para el entorno de ejecución) se encarga el propio entorno de ejecución de Java. Es decir, que por el hecho de ejecutar un método constructor, el entorno sabrá que tiene que realizar una serie de tareas (solicitud de una zona de memoria disponible, reserva de memoria para los atributos, enlace de la variable `objeto` a esa zona, etc.) y se pondrá rápidamente a desempeñarlas.

Cuando escribas el código de una clase no es necesario que implementes el método constructor si no quieres hacerlo. Java se encarga de dotar de un constructor por omisión (también conocido como constructor por defecto) a toda clase. Ese constructor por omisión se ocupará exclusivamente de las tareas de reserva de memoria. Si deseas que el constructor realice otras tareas adicionales, tendrás que escribirlo tú. El constructor por omisión no tiene parámetros.

#### Acuerda

El constructor por defecto no se ve en el código de una clase. Lo incluirá el compilador de Java al compilar la clase si descubre que no se ha creado ningún método constructor para esa clase.

Algunos ejemplos de instanciación o creación de objetos podrían ser:

```
1 p1 = new Punto();
2 r1 = new Rectangulo();
3 r2 = new Rectangulo();
4 cocheAntonio = new Coche();
5 palabra = new String(); //palabra = new String("");
```

#### Importante

En el caso de los constructores, si éstos no tienen parámetros, pueden omitirse los paréntesis vacíos.

Un objeto puede ser declarado e instanciado en la misma línea. Por ejemplo:

```
1 Punto p1 = new Punto();
```

### 16.6.3. Manipulación de un objeto: utilización de métodos y atributos.

Una vez que un objeto ha sido declarado y creado (clase instanciada) ya sí se puede decir que el objeto existe en el entorno de ejecución, y por tanto que puede ser manipulado como un objeto más en el programa, haciéndose uso de sus atributos y sus métodos.

Para acceder a un miembro de un objeto se utiliza el operador punto ( . ) del siguiente modo:

```
1 <nombreObjeto>.<nombreMiembro>
```

Donde `<nombreMiembro>` será el nombre de algún miembro del objeto (atributo o método) al cual se tenga acceso.

Por ejemplo, en el caso de los objetos de tipo `Punto` que has declarado e instanciado en los apartados anteriores, podrías acceder a sus miembros de la siguiente manera:

```
1 Punto p1, p2, p3;
2
3 p1= new Punto();
4 p1.x= 5;
5 p1.y= 6;
6
7 System.out.printf ("p1.x: %d\np1.y: %d\n", p1.x, p1.y);
8 System.out.printf ("p1.x: %d\np1.y: %d\n", p1.obtenerX(), p1.obtenerY());
9 p1.establecerX(25);
10 p1.establecerX(30);
11 System.out.printf ("p1.x: %d\np1.y: %d\n", p1.obtenerX(), p1.obtenerY());
```

Es decir, colocando el operador punto ( . ) a continuación del nombre del objeto y seguido del nombre del miembro al que se desea acceder.

## 16.7. Constructores.

Como ya has estudiado en unidades anteriores, en el ciclo de vida de un objeto se pueden distinguir las fases de:

- Construcción del objeto.
- Manipulación y utilización del objeto accediendo a sus miembros.
- Destrucción del objeto.

Como has visto en el apartado anterior, durante la fase de construcción o instanciación de un objeto es cuando se reserva espacio en memoria para sus atributos y se inicializan algunos de ellos. Un constructor es un método especial con el mismo nombre de la clase y que se encarga de realizar este proceso.

El proceso de declaración y creación de un objeto mediante el operador `new` ya ha sido estudiado en apartados anteriores. Sin embargo las clases que hasta ahora has creado no tenían constructor. Has estado utilizando los constructores por defecto que proporciona Java al compilar la clase. Ha llegado el momento de que empieces a implementar tus propios constructores.

### Información

Los métodos constructores se encargan de llevar a cabo el proceso de creación o construcción de un objeto.

#### 16.7.1. Concepto de constructor.

Un constructor es un método que tiene el mismo nombre que la clase a la que pertenece y que no devuelve ningún valor tras su ejecución. Su función es la de proporcionar el mecanismo de creación de instancias (objetos) de la clase.

Cuando un objeto es declarado, en realidad aún no existe. Tan solo se trata de un nombre simbólico (una variable) que en el futuro hará referencia a una zona de memoria que contendrá la información que representa realmente a un objeto. Para que esa variable de objeto aún "vacía" (se suele decir que es una referencia nula o vacía) apunte, o haga referencia a una zona de memoria que represente a una instancia de clase (objeto) existente, es necesario "construir" el objeto. Ese proceso se realizará a través del método constructor de la clase. Por tanto para crear un nuevo objeto es necesario realizar una llamada a un método constructor de la clase a la que pertenece ese objeto.

Ese proceso se realiza mediante la utilización del operador `new`.

Hasta el momento ya has utilizado en numerosas ocasiones el operador `new` para instanciar o crear objetos. En realidad lo que estabas haciendo era una llamada al constructor de la clase para que reservara memoria para ese objeto y por tanto "crear" físicamente el objeto en la memoria (dotarlo de existencia física dentro de la memoria del ordenador). Dado que en esta unidad estás ya definiendo tus propias clases, parece que ha llegado el momento de que empieces a escribir también los constructores de tus clases.

Por otro lado, si un constructor es al fin y al cabo una especie de método (aunque algo especial) y Java soporta la sobrecarga de métodos, podrías plantearte la siguiente pregunta: ¿podrá una clase disponer de más de constructor? En otras palabras, ¿será posible la sobrecarga de constructores? La respuesta es afirmativa.

### Importante

Una misma clase puede disponer de varios constructores. **Los constructores soportan la sobrecarga.**

Es necesario que toda clase tenga al menos un constructor. Si no se define ningún constructor en una clase, el compilador creará por nosotros un constructor por defecto vacío que se encarga de inicializar todos los atributos a sus valores por defecto (0 para los numéricos, null para las referencias, false para los boolean, etc.).

Algunas analogías que podrías imaginar para representar el constructor de una clase podrían ser:

- Los moldes de cocina para flanes, galletas, pastas, etc.
- Un cubo de playa para crear castillos de arena.
- Un molde de un lingote de oro.
- Una bolsa para hacer cubitos de hielo.

Una vez que incluyas un constructor personalizado a una clase, el compilador ya no incluirá el constructor por defecto (sin parámetros) y por tanto si intentas usarlo se produciría un error de compilación. Si quieres que tu clase tenga también un constructor sin parámetros tendrás que escribir su código (ya no lo hará por ti el compilador)

#### 16.7.2. Creación de constructores.

Cuando se escribe el código de una clase normalmente se pretende que los objetos de esa clase se creen de una determinada manera. Para ello se definen uno o más constructores en la clase. En la definición de un constructor se indican:

- El tipo de acceso.
- El nombre de la clase (el nombre de un método constructor es siempre el nombre de la propia clase).
- La lista de parámetros que puede aceptar.
- Si lanza o no excepciones.
- El cuerpo del constructor (un bloque de código como el de cualquier método).

Como puedes observar, la estructura de los constructores es similar a la de cualquier método, con las excepciones de que no tiene tipo de dato devuelto (no devuelve ningún valor) y que el nombre del método constructor debe ser obligatoriamente el nombre de la clase.

### Recuerda

Si defines constructores personalizados para una clase, el constructor por defecto (sin parámetros) para esa clase deja de ser generado por el compilador, de manera que tendrás que crearlo tú si quieras poder utilizarlo.

Si se ha creado un constructor con parámetros y no se ha implementado el constructor por defecto, el intento de utilización del constructor por defecto producirá un error de compilación (el compilador no lo hará por nosotros).

Un ejemplo de constructor para la clase Punto podría ser:

```
1 public Punto(int x, int y) {
2 this.x=x;
3 this.y=y;
4 cantidadPuntos++; // Suponiendo que tengamos un atributo estático cantidadPuntos
5 }
```

En este caso el constructor recibe dos parámetros. Además de reservar espacio para los atributos (de lo cual se encarga automáticamente Java), también asigna sendos valores iniciales a los atributos `x` e `y`. Por último incrementa un atributo (probablemente estático) llamado `cantidadPuntos`.

#### 16.7.3. Utilización de constructores.

Una vez que dispongas de tus propios constructores personalizados, la forma de utilizarlos es igual que con el constructor por defecto (mediante la utilización de la palabra reservada `new`) pero teniendo en cuenta que si has declarado parámetros en tu

método constructor, tendrás que llamar al constructor con algún valor para esos parámetros. Un ejemplo de utilización del constructor que has creado para la clase `Punto` en el apartado anterior podría ser:

```
1 Punto p1;
2 p1= new Punto(10, 7);
```

En este caso no se estaría utilizando el constructor por defecto sino el constructor que acabas de implementar en el cual además de reservar memoria se asigna un valor a algunos de los atributos.

#### 16.7.4. Constructores de copia. [NUEVO]

Una forma de iniciar un objeto podría ser mediante la copia de los valores de los atributos de otro objeto ya existente. Imagina que necesitas varios objetos iguales (con los mismos valores en sus atributos) y que ya tienes uno de ellos perfectamente configurado (sus atributos contienen los valores que tú necesitas). Estaría bien disponer de un constructor que hiciera copias idénticas de ese objeto.

Durante el proceso de creación de un objeto puedes generar objetos exactamente iguales(basados en la misma clase) que se distinguirán posteriormente porque podrán tener estados distintos (valores diferentes en los atributos). La idea es poder decirle a la clase que además de generar un objeto nuevo, que lo haga con los mismos valores que tenga otro objeto ya existente. Es decir, algo así como si pudieras clonar el objeto tantas veces como te haga falta. A este tipo de mecanismo se le suele llamar constructor copia o constructor de copia.

Un constructor copia es un método constructor como los que ya has utilizado pero con la particularidad de que recibe como parámetro una referencia al objeto cuyo contenido se desea copiar. Este método revisa cada uno de los atributos del objeto recibido como parámetro y se copian todos sus valores en los atributos del objeto que se está creando en ese momento en el método constructor.

Un ejemplo de constructor copia para la clase `Punto` podría ser:

```
1 public Punto(Punto p){
2 this.x = p.obtenerX();
3 this.y = p.obtenerY();
4 }
```

En este caso el constructor recibe como parámetro un objeto del mismo tipo que el que va a ser creado (clase `Punto`), inspecciona el valor de sus atributos (atributos `x` e `y`), y los reproduce en los atributos del objeto en proceso de construcción (`this`).

Un ejemplo de utilización de ese constructor podría ser:

```
1 Punto p1, p2;
2 p1 = new Punto (10, 7);
3 p2 = new Punto (p1);
```

En este caso el objeto `p2` se crea a partir de los valores del objeto `p1`.

#### 16.7.5. Destrucción de objetos.

Como ya has estudiado en unidades anteriores, cuando un objeto deja de ser utilizado, los recursos usados por él (memoria, acceso a archivos, conexiones con bases de datos, etc.) deberían de ser liberados para que puedan volver a ser utilizados por otros procesos (mecanismo de destrucción del objeto).

Mientras que de la construcción de los objetos se encargan los métodos constructores, de la destrucción se encarga un proceso del entorno de ejecución conocido como **recolector de basura (garbage collector)**. Este proceso va buscando periódicamente objetos que ya no son referenciados (no hay ninguna variable que haga referencia a ellos) y los marca para ser eliminados. Posteriormente los irá eliminando de la memoria cuando lo considere oportuno (en función de la carga del sistema, los recursos disponibles, etc.).

Normalmente se suele decir que en Java no hay método destructor y que en otros lenguajes orientados a objetos como C++, sí se implementa explícitamente el destructor de una clase de la misma manera que se define el constructor. En realidad en Java también es posible implementar el método destructor de una clase, se trata del método `finalize()`.

Este método `finalize` es llamado por el recolector de basura cuando va a destruir el objeto (lo cual nunca se sabe cuándo va a suceder exactamente, pues una cosa es que el objeto sea marcado para ser borrado y otra que sea borrado efectivamente). Si ese método no existe, se ejecutará un destructor por defecto (el método `finalize` que contiene la clase `Object`, de la cual heredan todas las clases en Java) que liberará la memoria ocupada por el objeto. Se recomienda por tanto que si un objeto utiliza determinados recursos de los cuales no tienes garantía que el entorno de ejecución los vaya a liberar (cerrar archivos, cerrar

conexiones de red, cerrar conexiones con bases de datos, etc.), implementes explícitamente un método `finalize` en tus clases. Si el único recurso que utiliza tu clase es la memoria necesaria para albergar sus atributos, eso sí será liberado sin problemas. Pero si se trata de algo más complejo, será mejor que te encargues tú mismo de hacerlo implementando tu destructor personalizado (`finalize`).

Por otro lado, esta forma de funcionar del entorno de ejecución de Java (destrucción de objetos no referenciados mediante el recolector de basura) implica que no puedas saber exactamente cuándo un objeto va a ser definitivamente destruido, pues si una variable deja de ser referenciada (se cierra el ámbito de ejecución donde fue creada) no implica necesariamente que sea inmediatamente borrada, sino que simplemente es marcada para que el recolector la borre cuando pueda hacerlo.

Si en un momento dado fuera necesario garantizar que el proceso de finalización (método `finalize`) sea invocado, puedes recurrir al método `runFinalization()` de la clase `System` para forzarlo:

```
1 System.runFinalization();
```

Este método se encarga de llamar a todos los métodos `finalize` de todos los objetos marcados por el recolector de basura para ser destruidos.

Si necesitas implementar un destructor (normalmente no será necesario), debes tener en cuenta que:

- El nombre del método destructor debe ser `finalize()`.
- No puede recibir parámetros.
- Sólo puede haber un destructor en una clase. No es posible la sobrecarga dado que no tiene parámetros.
- No puede devolver ningún valor. Debe ser de tipo `void`.

## 16.8. Clases Anidadas, Clases Internas (Inner Class) [NUEVO]

### 16.8.1. Clase Anidada (Nested Class):

Una clase anidada es una clase estática definida dentro de otra clase. Se declara con el modificador `static`.

Asociación:

- No tiene una relación directa con las instancias de la clase externa.
- Se puede acceder a ella sin necesidad de crear una instancia de la clase externa.

Contexto:

- Solo puede acceder a los miembros `static` de la clase externa.
- No puede acceder a miembros de instancia (no estáticos) de la clase externa.

Uso común:

- Se utiliza cuando la clase anidada tiene un propósito independiente de las instancias de la clase externa.

#### Ejemplo:

```
1 public class Empresa {
2 [...]
3 static class PlantillaCompletaException extends Exception {
4 public PlantillaCompletaException() {
5 super("La empresa tiene la plantilla completa.");
6 }
7 }
8 }
```

Así después podemos usarla en el `catch` accediendo directamente a la clase `Empresa` (ya que es `static`):

```
1 public class TestEmpresa {
2 public static void main(String[] args) {
3 [...]
4 try {
5 [...]
6 } catch (Empresa.PlantillaCompletaException ex) {
7 System.err.println(ex.getMessage());
8 }
9 [...]
10 }
11 }
```

### 16.8.2. Clase Interna (Inner Class):

Una clase interna es una clase no estática definida dentro de otra clase. No se declara con `static`.

Asociación:

- Está directamente asociada con una instancia de la clase externa.
- Para instanciarla, primero necesitas una instancia de la clase externa.

Contexto:

- Puede acceder a **todos** los miembros (estáticos y no estáticos) de la clase externa, incluso si son privados.
- La clase interna tiene una referencia implícita a la instancia de la clase externa.

Uso común:

- Se utiliza cuando la clase interna necesita interactuar con los miembros de instancia de la clase externa.

#### Ejemplo:

```

1 class Pc {
2
3 double precio;
4
5 public String toString() {
6 return "El precio del PC es " + this.precio;
7 }
8
9 class Monitor {
10
11 String marca;
12
13 public String toString() {
14 return "El monitor es de la marca " + this.marca;
15 }
16 }
17
18 class Cpu {
19
20 String marca;
21
22 public String toString() {
23 return "La CPU es de la marca " + this.marca;
24 }
25 }
26 }
27
28 public class ClaseInternaHardware {
29
30 public static void main(String[] args) {
31 Pc miPc = new Pc();
32 Pc.Monitor miMonitor = miPc.new Monitor();
33 Pc.Cpu miCpu = miPc.new Cpu();
34 miPc.precio = 1250.75;
35 miMonitor.marca = "Asus";
36 miCpu.marca = "AMD";
37 System.out.println(miPc); //El precio del PC es 1250.75
38 System.out.println(miMonitor); //El monitor es de la marca Asus
39 System.out.println(miCpu); //La CPU es de la marca AMD
40 }
41 }
```

### 16.8.3. Comparación Resumida:

| Característica                      | Clase Anidada (Nested)                                                     | Clase Interna (Inner)                                              |
|-------------------------------------|----------------------------------------------------------------------------|--------------------------------------------------------------------|
| <b>Es estática</b>                  | Sí ( <code>static</code> )                                                 | No ( <code>non-static</code> )                                     |
| <b>Acceso a clase externa</b>       | Solo miembros <code>static</code>                                          | Miembros estáticos y no estáticos                                  |
| <b>Asociación con clase externa</b> | No está asociada a instancias                                              | Está asociada a una instancia de la clase externa                  |
| <b>Uso común</b>                    | Lógica independiente de instancias de la clase externa                     | Lógica que necesita interactuar con la instancia externa           |
| <b>Instanciación</b>                | <code>OuterClass.NestedClass nested = new OuterClass.NestedClass();</code> | <code>OuterClass.InnerClass inner = outer.new InnerClass();</code> |

#### 16.8.4. Convenciones comunes sobre el número de clases en un archivo:

1. **Una clase pública por archivo:** Es una convención estándar en Java que cada archivo .java debe contener solo una clase pública, y el nombre del archivo debe coincidir exactamente con el nombre de la clase pública (incluyendo mayúsculas y minúsculas). Esto es obligatorio para que el compilador de Java pueda encontrar las clases correctamente.

Ejemplo:

```
1 // Archivo: MiClase.java
2 public class MiClase {
3 // Código de la clase
4 }
```

#### 2. Clases no públicas en el mismo archivo:

Puedes incluir múltiples clases no públicas (con modificador default o package-private) en el mismo archivo, siempre que no haya más de una clase pública.

Ejemplo:

```
1 // Archivo: ClasePrincipal.java
2 public class ClasePrincipal {
3 // Código de la clase pública
4 }
5
6 class ClaseAuxiliar {
7 // Código de la clase auxiliar
8 }
```

Sin embargo, esto no es una práctica común en proyectos grandes, ya que puede dificultar el mantenimiento y la localización de clases.

#### 2. Separar clases auxiliares en archivos propios:

Aunque no es obligatorio, es una buena práctica colocar cada clase en su propio archivo, incluso si no es pública. Esto facilita:

- La localización y lectura de la clase.
- El control de versiones y la fusión de cambios en sistemas de control de código fuente.
- La prueba unitaria y el mantenimiento.

#### 1. Uso de clases anidadas:

Si una clase es relevante solo dentro del contexto de otra clase, considera usar una clase anidada o una clase interna en lugar de una clase separada.

Ejemplo:

```
1 ````java
2 public class ClaseExterna {
3 private static class ClaseAnidada {
4 // Clase anidada
5 }
6
7 private class InnerClass {
8 // Clase interna
9 }
10 }
11 ````
```

#### 1. Evitar clases no relacionadas en un solo archivo:

Colocar múltiples clases no relacionadas en un solo archivo se considera una mala práctica, ya que:

- Puede hacer que el archivo sea difícil de leer.
- Introduce acoplamiento innecesario entre clases.
- Viola el principio de responsabilidad única (SRP).

#### 1. Clases utilitarias:

Para clases con métodos estáticos (por ejemplo, `utils` o `Constants`), es común colocarlas en un único archivo, ya que suelen ser pequeñas y no tienen lógica compleja.

## 16.9. Introducción a la herencia. [NUEVO]

La herencia es uno de los conceptos fundamentales que introduce la programación orientada a objetos. La idea fundamental es permitir crear nuevas clases aprovechando las características (atributos y métodos) de otras clases ya creadas evitando así tener que volver a definir esas características (reutilización).

A una clase que hereda de otra se le llama subclase o clase hija y aquella de la que se hereda es conocida como superclase o clase padre. También se puede hablar en general de clases descendientes o clases ascendientes. Al heredar, la subclase adquiere todas las características (atributos y métodos) de su superclase, aunque algunas de ellas pueden ser sobreescritas o modificadas dentro de la subclase (a eso se le suele llamar **especialización**).

Una clase puede heredar de otra que a su vez ha podido heredar de una tercera y así sucesivamente. Esto significa que las clases van tomando todas las características de sus clases ascendientes (no sólo de su superclase o clase padre inmediata) a lo largo de toda la rama del árbol de la jerarquía de clases en la que se encuentre.

Imagina que quieras modelar el funcionamiento de algunos vehículos para trabajar con ellos en un programa de simulación. Lo primero que haces es pensar en una clase `Vehículo` que tendrá un conjunto de atributos (por ejemplo: posición actual, velocidad actual y velocidad máxima que puede alcanzar el vehículo) y de métodos (por ejemplo: detener, acelerar, frenar, establecer dirección, establecer sentido).

Dado que vas a trabajar con muchos tipos de vehículos, no tendrás suficiente con esas características, así que seguramente vas a necesitar nuevas clases que las incorporen. Pero las características básicas que has definido en la clase `Vehículo` van a ser compartidas por cualquier nuevo vehículo que vayas a modelar. Esto significa que si creas otra clase podrías heredar de `Vehículo` todas esos atributos y propiedades y tan solo tendrías que añadir las nuevas.

Si vas a trabajar con vehículos que se desplazan por tierra, agua y aire, tendrás que idear nuevas clases con características adicionales. Por ejemplo, podrías crear una clase `VehiculoTerrestre`, que herede las características de `Vehículo`, pero que también incorpore atributos como el número de ruedas o la altura de los bajos). A su vez, podría idearse una nueva clase que herede de `VehiculoTerrestre` y que incorpore nuevos atributos y métodos como, por ejemplo, una clase `Coche`. Y así sucesivamente con toda la jerarquía de clases heredadas que consideres oportunas para representar lo mejor posible el entorno y la información sobre la que van a trabajar tus programas.

### 16.9.1. Creación y utilización de clases heredadas.

¿Cómo se indica en Java que una clase hereda de otra? Para indicar que una clase hereda de otra es necesario utilizar la palabra reservada `extends` junto con el nombre de la clase de la que se quieren heredar sus características:

```
1 class<NombreClase> extends <nombreSuperClase> {
2 ...
3 }
```

En el ejemplo anterior de los vehículos, la clase `VehiculoTerrestre` podría quedar así al ser declarada:

```
1 class VehiculoTerrestre extends Vehiculo {
2 ...
3 }
```

Y en el caso de la clase `Coche`:

```
1 class Coche extends VehiculoTerrestre {
2 ...
3 }
```

En unidades posteriores estudiarás detalladamente cómo crear una jerarquía de clases y qué relación existe entre la herencia y los distintos modificadores de clases, atributos y métodos. Por ahora es suficiente con que entiendas el concepto de herencia y sepas reconocer cuándo una clase hereda de otra (uso de la palabra reservada `extends`).

Puedes comprobar que en las bibliotecas proporcionadas por Java aparecen jerarquías bastante complejas de clases heredadas en las cuales se han ido aprovechando cada uno de los miembros de una clase base para ir construyendo las distintas clases derivadas añadiendo (y a veces modificando) poco a poco nueva funcionalidad.

Eso suele suceder en cualquier proyecto de software conforme se van a analizando, descomponiendo y modelando los datos con los que hay que trabajar. La idea es poder representar de una manera eficiente toda la información que es manipulada por el sistema que se desea automatizar. Una jerarquía de clases suele ser una buena forma de hacerlo.

En el caso de Java, cualquier clase con la que trabajes tendrá un ascendiente. Si en la declaración de clase no indicas la clase de la que se hereda (no se incluye un `extends`), el compilador considerará automáticamente que se hereda de la clase `Object`, que es la clase que se encuentra en el nivel superior de toda la jerarquía de clases en Java (y que es la única que no hereda de nadie).

También irás viendo al estudiar distintos componentes de las bibliotecas de Java (por ejemplo en el caso de las interfaces gráficas) que para poder crear objetos basados en las clases proporcionadas por esas bibliotecas tendrás que crear tus propias clases que hereden de algunas de esas clases. Para ellos tendrás que hacer uso de la palabra reservada `extends`.

### Recuerda

En Java todas las clases son descendientes (de manera explícita o implícita) de la clase `Object`.

## 16.10. Conversión entre objetos (Casting) [NUEVO]

La esencia de Casting permite convertir un dato de tipo primitivo en otro generalmente de más precisión.

Entre objetos es posible realizar el casting. Tenemos una clase persona con una subclase empleado y este a su vez una subclase encargado.

```
classDiagram
Persona <|-- Empleado
Empleado <|-- Encargado
```

Si creamos una instancia de tipo persona y le asignamos un objeto de tipo empleado o encargado, al ser una subclase no existe ningún tipo de problema, ya que todo encargado o empleado es persona.

Por otro lado, si intentamos asignar valores a los atributos específicos de empleado o encargado nos encontramos con una pérdida de precisión puesto que no se pueden ejecutar todos los métodos de los que dispone un objeto de tipo empleado o encargado, ya que persona contiene menos métodos que la clase empleado o encargado. En este caso es necesario hacer un casting, sino el compilador dará error.

Ejemplo:

```
1 package UD05;
2
3 // Clase Persona que solo dispone de nombre
4 public class Persona {
5
6 String nombre;
7
8 public Persona(String nombre) {
9 this.nombre = nombre;
10 }
11
12 public void setNombre(String nom) {
13 nombre = nom;
14 }
15
16 public String getNombre() {
17 return nombre;
18 }
19
20 @Override
21 public String toString() {
22 return "Nombre: " + nombre;
23 }
24 }
```

```

1 package UD05;
2
3 // Clase Empleado que hereda de Persona y añade atributo sueldoBase
4 public class Empleado extends Persona {
5
6 double sueldoBase;
7
8 public Empleado(String nombre, double sueldoBase) {
9 super(nombre);
10 this.sueldoBase = sueldoBase;
11 }
12
13 public double getSueldo() {
14 return sueldoBase;
15 }
16
17 public void setSueldoBase(double sueldoBase) {
18 this.sueldoBase = sueldoBase;
19 }
20
21 @Override
22 public String toString() {
23 return super.toString() + "\nSueldo Base: " + sueldoBase;
24 }
25 }
```

```

1 package UD05;
2
3 // Clase Encargado que hereda de Empleado y añade atributo seccion
4 public class Encargado extends Empleado {
5
6 String seccion;
7
8 public Encargado(String nombre, double sueldoBase, String seccion) {
9 super(nombre, sueldoBase);
10 this.seccion = seccion;
11 }
12
13 public String getSeccion() {
14 return seccion;
15 }
16
17 public void setSeccion(String seccion) {
18 this.seccion = seccion;
19 }
20
21 @Override
22 public String toString() {
23 return super.toString() + "\nSección:" + seccion ;
24 }
25 }
```

```

1 package UD05;
2
3 public class Casting {
4
5 public static void main(String[] args) {
6 // Casting Implicito
7 Persona encargadoCarniceria = new Encargado("Rosa Ramos", 1200, "Carniceria");
8
9 // No tenemos disponibles los métodos de la clase Encargado:
10 //EncargadaCarniceria.setSueldoBase(1200);
11 //EncargadaCarniceria.setSeccion("Carniceria");
12 //Pero al imprimir se imprime con el método más específico (luego lo vemos)
13 System.out.println(encargadoCarniceria);
14
15 // Casting Explicito
16 Encargado miEncargado = (Encargado) encargadoCarniceria;
17 //Tenemos disponibles los métodos de la clase Encargado:
18 miEncargado.setSueldoBase(1200);
19 miEncargado.setSeccion("Carniceria");
20 //Al imprimir se imprime con el método más específico de nuevo.
21 System.out.println(miEncargado);
22 }
23 }
```

Las reglas a la hora de realizar casting es que:

- cuando se utiliza una clase más específicas (más abajo en la jerarquía) no hace falta casting. Es lo que llamamos **casting implícito**.
- cuando se utiliza una clase menos específica (más arriba en la jerarquía) hay que hacer un **casting explícito**.



### Porqué a la hora de imprimir el casting implicito la clase más genérica se imprime con el método más especializado?

Debes entender que en realidad `encargadoCarniceria` es un `Encargado` que se *disfraz*a de `Persona`, pero en realidad sus métodos son los especializados (el `toString()` más moderno sobrescribe al de sus padres. Recuerda que la anotación `@override` es opcional, y aunque no se indique el método sigue sobrescribiendo al de su padre)

Si por ejemplo usamos este fragmento:

```
1 //Persona
2 Persona David = new Persona ("David");
3 System.out.println(David);
```

Se imprimirá con el método `toString()` de la clase `Persona` (sólo el nombre).

Y si hacemos un casting del objeto `David` a uno más genérico (`Object`) seguirá usando el método más especializado:

```
1 //Object
2 Object oDavid = David;
3 System.out.println(oDavid);
```

## 16.11. Acceso a métodos de la superclase [NUEVO]

Para acceder a los métodos de la superclase se utiliza la sentencia `super`. La sentencia `this` permite acceder a los campos y métodos de la clase. La sentencia `super` permite acceder a los campos y métodos de la superclase. El uso de `super` lo hemos visto en las clases `Empleado` y `Encargado` anteriores:

```
1 public class Persona {
2 private String nombre;
3
4 public Persona(String nombre){
5 this.nombre=nombre;
6 }
7 }
```

```
1 public class Empleado extends Persona {
2 [...]
3 public Empleado(String nombre, double sueldoBase) {
4 super(nombre);
5 this.sueldoBase = sueldoBase;
6 }
7 [...]
```

```
1 public class Encargado extends Empleado {
2 [...]
3 public Encargado(String nombre, double sueldoBase, String seccion) {
4 super(nombre, sueldoBase);
5 this.seccion = seccion;
6 }
7 [...]
```

Podemos mostrar el nombre de la clase y el nombre de la clase de la que hereda con `getClass()` y `getSuperclass()`. Ejemplo:

```
1 package UD05;
2
3 public class Anexo04SuperClase {
4
5 public static void main(String[] args) {
6 Empleado empleadoCarniceria = new Empleado("Rosa Ramos", 1200);
7 // Muestra los datos del Empleado
8 System.out.println(empleadoCarniceria instanceof Encargado); //false
9 System.out.println(empleadoCarniceria.getClass()); //class Empleado
10 System.out.println(empleadoCarniceria.getClass().getSuperclass()); //class Persona
11 }
12 }
```

## 16.12. Empaquetado de clases [NUEVO]

La encapsulación de la información dentro de las clases ha permitido llevar a cabo el proceso de ocultación, que es fundamental para el trabajo con clases y objetos. Es posible que conforme vaya aumentando la complejidad de tus aplicaciones necesites que algunas de tus clases puedan tener acceso a parte de la implementación de otras debido a las relaciones que se establezcan

entre ellas a la hora de diseñar tu modelo de datos. En estos casos se puede hablar de un nivel superior de encapsulamiento y ocultación conocido como empaquetado.

Un paquete consiste en un conjunto de clases relacionadas entre sí y agrupadas bajo un mismo nombre. Normalmente se encuentran en un mismo paquete todas aquellas clases que forman una biblioteca o que reúnen algún tipo de característica en común. Esto la organización de las clases para luego localizar fácilmente aquellas que vayas necesitando.

#### **16.12.1. Jerarquía de paquetes.**

Los paquetes en Java pueden organizarse jerárquicamente de manera similar a lo que puedes encontrar en la estructura de carpetas en un dispositivo de almacenamiento, donde:

- Las clases serían como los archivos.
- Cada paquete sería como una carpeta que contiene archivos (clases).
- Cada paquete puede además contener otros paquetes (como las carpetas que contienen carpetas).
- Para poder hacer referencia a una clase dentro de una estructura de paquetes, habrá que indicar la trayectoria completa desde el paquete raíz de la jerarquía hasta el paquete en el que se encuentra la clase, indicando por último el nombre de la clase (como el path absoluto de un archivo).

La estructura de paquetes en Java permite organizar y clasificar las clases, evitando conflictos de nombres y facilitando la ubicación de una clase dentro de una estructura jerárquica.

Por otro lado, la organización en paquetes permite también el control de acceso a miembros de las clases desde otras clases que estén en el mismo paquete gracias a los modificadores de acceso (recuerda que uno de los modificadores que viste era precisamente el de paquete).

Las clases que forman parte de la jerarquía de clases de Java se encuentran organizadas en diversos paquetes.

Todas las clases proporcionadas por Java en sus bibliotecas son miembros de distintos paquetes y se encuentran organizadas jerárquicamente. Dentro de cada paquete habrá un conjunto de clases con algún tipo de relación entre ellas. Se dice que todo ese conjunto de paquetes forman la API de Java. Por ejemplo las clases básicas del lenguaje se encuentran en el paquete `java.lang`, las clases de entrada/salida las podrás encontrar en el paquete `java.io` y en el paquete `java.math` podrás observar algunas clases para trabajar con números grandes y de gran precisión.

#### **16.12.2. Utilización de los paquetes.**

Es posible acceder a cualquier clase de cualquier paquete (siempre que ese paquete esté disponible en nuestro sistema, obviamente) mediante la calificación completa de la clase dentro de la estructura jerárquica de paquete. Es decir indicando la trayectoria completa de paquetes desde el paquete raíz hasta la propia clase. Eso se puede hacer utilizando el operador punto (`.`) para especificar cada subpaquete:

```
1 paquete_raiz.subpaquete1.subpaquete2.subpaquete_n.NombreClase
```

Por ejemplo:

```
1 java.lang.String
```

En este caso se está haciendo referencia a la clase `String` que se encuentra dentro del paquete `java.lang`. Este paquete contiene las clases elementales para poder desarrollar una aplicación Java.

Otro ejemplo podría ser:

```
1 java.util.regex.Patern
```

En este otro caso se hace referencia a la clase `Patern` ubicada en el paquete `java.util.regex`, que contiene clases para trabajar con expresiones regulares.

Dado que puede resultar bastante tedioso tener que escribir la trayectoria completa de una clase cada vez que se quiera utilizar, existe la posibilidad de indicar que se desea trabajar con las clases de uno o varios paquetes. De esa manera cuando se vaya a utilizar una clase que pertenezca a uno de esos paquetes no será necesario indicar toda su trayectoria. Para ello se utiliza la sentencia `import` (importar):

```
1 import paquete_raiz.subpaquete1.subpaquete2.subpaquete_n.NombreClase;
```

De esta manera a partir de ese momento podrá utilizarse directamente `NOMBRECLASE` en lugar de toda su trayectoria completa.

Los ejemplos anteriores quedarían entonces:

```
1 import java.lang.String;
2 import java.util.regex.Patern;
```

Si suponemos que vamos a utilizar varias clases de un mismo paquete, en lugar de hacer un `import` de cada una de ellas, podemos utilizar el comodín (símbolo asterisco: `*`) para indicar que queremos importar todas las clases de ese paquete y no sólo una determinada:

```
1 import java.lang.*;
2 import java.util.regex.*;
```

Si un paquete contiene subpaquetes, el comodín no importará las clases de los subpaquetes, tan solo las que haya en el paquete. La importación de las clases contenidas en los subpaquetes habrá que indicarla explícitamente. Por ejemplo:

```
1 import java.util.*;
2 import java.util.regex.*;
```

En este caso se importarán todas las clases del paquete `java.util` (clases `Date`, `Calendar`, `Timer`, etc.) y de su subpaquete `java.util.regex` (`Matcher` y `Pattern`), pero no las de otros subpaquetes como `java.util.concurrent` o `java.util.jar`. Por último tan solo indicar que en el caso del paquete `java.lang`, no es necesario realizar importación. El compilador, dada la importancia de este paquete, permite el uso de sus clases sin necesidad de indicar su trayectoria (es como si todo archivo Java incluyera en su primera línea la sentencia `import java.lang.*`).

#### 16.12.3. Inclusión de una clase en un paquete.

Al principio de cada archivo `.java` se puede indicar a qué paquete pertenece mediante la palabra reservada `package` seguida del nombre del paquete:

```
1 package nombre_paquete;
```

Por ejemplo:

```
1 package paqueteEjemplo;
2 class claseEjemplo {
3 ...
4 }
```

La sentencia `package` debe ser incluida en cada archivo fuente de cada clase que quieras incluir ese paquete. Si en un archivo fuente hay definidas más de una clase, todas esas clases formarán parte del paquete indicado en la sentencia `package`.

Si al comienzo de un archivo Java no se incluyen ninguna sentencia `package`, el compilador considerará que las clases de ese archivo formarán parte del paquete por omisión (un paquete sin nombre asociado al proyecto).

Para evitar la ambigüedad, dentro de un mismo paquete no puede haber dos clases con el mismo nombre, aunque sí pueden existir clases con el mismo nombre si están en paquetes diferentes. El compilador será capaz de distinguir una clase de otra gracias a que pertenecen a paquetes distintos.

Como ya has visto en unidades anteriores, el nombre de un archivo fuente en Java se construye utilizando el nombre de la clase pública que contiene junto con la extensión `.java`, pudiendo haber únicamente una clase pública por cada archivo fuente. El nombre de la clase debía coincidir (en mayúsculas y minúsculas) exactamente con el nombre del archivo en el que se encontraba definida.

Así, si por ejemplo tenías una clase `Punto` dentro de un archivo `Punto.java`, la compilación daría lugar a un archivo `Punto.class`.

En el caso de los paquetes, la correspondencia es a nivel de directorios o carpetas. Es decir, si la clase `Punto` se encuentra dentro del paquete `prog.figuras`, el archivo `Punto.java` debería encontrarse en la carpeta `prog\figuras`. Para que esto funcione correctamente el compilador ha de ser capaz de localizar todos los paquetes (tanto los estándar de Java como los definidos por otros programadores). Es decir, que el compilador debe tener conocimiento de dónde comienza la estructura de carpetas definida por los paquetes y en la cual se encuentran las clases. Para ello se utiliza el **ClassPath** cuyo funcionamiento habrás estudiado en las primeras unidades de este módulo. Se trata de una variable de entorno que contiene todas las rutas en las que comienzan las estructuras de directorios (distintas jerarquías posibles de paquetes) en las que están contenidas las clases.

#### 16.12.4. Proceso de creación de un paquete.

Para crear un paquete en Java te recomendamos seguir los siguientes pasos:

1. **Poner un nombre al paquete.** Suele ser habitual utilizar el dominio de Internet de la empresa que ha creado el paquete. Por ejemplo, para el caso de `miempresa.com`, podría utilizarse un nombre de paquete `com.miempresa`.
2. **Crear una estructura jerárquica de carpetas equivalente a la estructura de subpaquetes.** La ruta de la raíz de esa estructura jerárquica deberá estar especificada en el **ClassPath** de Java.
3. **Especificar a qué paquete pertenecen la clase (o clases) del archivo** `.java` mediante el uso de la sentencia package tal y como has visto en el apartado anterior.

Este proceso ya lo has debido de llevar a cabo en unidades anteriores al compilar y ejecutar clases con paquetes. Estos pasos simplemente son para que te sirvan como recordatorio del procedimiento que debes seguir a la hora de clasificar, jerarquizar y utilizar tus propias clases.

### 16.13. Ejercicios resueltos

#### 16.13.1. Modificadores de acceso

Imagina que quieres escribir una clase que represente un rectángulo en el plano. Para ello has pensado en los siguientes atributos:

- Atributos `x1`, `y1`, que representan la coordenadas del vértice inferior izquierdo del rectángulo. Ambos de tipo `double` (números reales).
- Atributos `x2`, `y2`, que representan las coordenadas del vértice superior derecho del rectángulo. También de tipo `double` (números reales).

Con estos dos puntos (`x1`, `y1`) y (`x2`, `y2`) se puede definir perfectamente la ubicación de un rectángulo en el plano.

Escribe una clase que contenga todos esos atributos teniendo en cuenta que queremos que sea una clase visible desde cualquier parte del programa y que sus atributos sean también accesibles desde cualquier parte del código.

**Respuesta:** Dado que se trata de una clase que podrá usarse desde cualquier parte del programa, utilizaremos el modificador de acceso `public` para la clase:

```
1 public class Rectangulo
```

Los cuatro atributos que necesitamos también han de ser visibles desde cualquier parte, así que también se utilizará el modificador de acceso `public` para los atributos:

```
1 public double x1, y1; // Vértice inferior izquierdo
2 public double x2, y2; // Vértice superior derecho
```

De esta manera la clase completa quedaría:

```
1 public class Rectangulo {
2 public double x1, y1; // Vértice inferior izquierdo
3 public double x2, y2; // Vértice superior derecho
4 }
```

#### 16.13.2. Atributos estáticos

Ampliar el ejercicio anterior del rectángulo incluyendo los siguientes atributos:

- Atributo `numRectangulos`, que almacena el número de objetos de tipo rectángulo creados hasta el momento.
- Atributo `nombre`, que almacena el nombre que se le quiera dar a cada rectángulo.
- Atributo `nombreFigura`, que almacena el nombre de la clase, es decir, "Rectángulo".
- Atributo `PI`, que contiene el nombre de la constante `PI` con una precisión de cuatro cifras decimales.

No se desea que los atributos `nombre` y `numRectangulos` puedan ser visibles desde fuera de la clase. Y además se desea que la clase sea accesible solamente desde su propio paquete.

**Respuesta:** Los atributos `numRectangulos`, `nombreFigura` y `PI` podrían ser estáticos pues se trata de valores más asociados a la propia clase que a cada uno de los objetos que se puedan ir creando. Además, en el caso de `PI` y `nombreFigura`, también podría

ser un atributo `final`, pues se trata de valores únicos y constantes (3.1416 en el caso de `PI` y "Rectángulo" en el caso de `nombreFigura`).

Dado que no se desea que se tenga accesibilidad a los atributos `nombre` y `numRectangulos` desde fuera de la clase podría utilizarse el atributo `private` para cada uno de ellos.

Por último hay que tener en cuenta que se desea que la clase sólo sea accesible desde el interior del paquete al que pertenece, por tanto habrá que utilizar el modificador por omisión o de paquete. Esto es, no incluir ningún modificador de acceso en la cabecera de la clase.

Teniendo en cuenta todo lo anterior la clase podría quedar finalmente así:

```

1 class Rectangulo { // Sin modificador "public" para que sólo sea accesible desde el paquete
2 // Atributos de clase
3 private static int numRectangulos; // Número total de rectángulos creados
4 public static final String NOMBREFIGURA = "Rectángulo"; // Nombre de la clase
5 public static final double PI = 3.1416; // Constante PI
6
7 // Atributos de objeto
8 private String nombre; // Nombre del rectángulo
9 public double x1, y1; // Vértice inferior izquierdo
10 public double x2, y2; // Vértice superior derecho
11 }
```

### 16.13.3. Cuerpo de un método

Vamos a seguir ampliando la clase en la que se representa un rectángulo en el plano (clase `Rectangulo`). Para ello has pensado en los siguientes métodos públicos:

- Métodos `getNombre` y `setNombre`, que permiten el acceso y modificación del atributo `nombre` del rectángulo.
- Método `calcularSuperficie`, que calcula el área encerrada por el rectángulo.
- Método `calcularPerimetro`, que calcula la longitud del perímetro del rectángulo.
- Método `desplazar`, que mueve la ubicación del rectángulo en el plano en una cantidad `x` (para el eje `X`) y otra cantidad `y` (para el eje `Y`). Se trata simplemente de sumar el desplazamiento `x` a las coordenadas `x1` y `x2`, y el desplazamiento `y` a las coordenadas `y1` e `y2`. Los parámetros de entrada de este método serán por tanto `x` e `y`, de tipo `double`.
- Método `obtenerNumRectangulos`, que devuelve el número de rectángulos creados hasta el momento.

Incluye la implementación de cada uno de esos métodos en la clase `Rectangulo`.

**Respuesta:** En el caso del método `obtenerNombre()`, se trata simplemente de devolver el valor del atributo `nombre`:

```

1 public String obtenerNombre () {
2 return nombre;
3 }
```

Para el implementar el método `establecerNombre` también es muy sencillo. Se trata de modificar el contenido del atributo `nombre` por el valor proporcionado a través de un parámetro de entrada:

```

1 public void establecerNombre (String nom) {
2 nombre = nom;
3 }
```

Los métodos de cálculo de `superficie` y `perímetro` no van a recibir ningún parámetro de entrada, tan solo deben realizar cálculos a partir de los atributos contenidos en el objeto para obtener los resultados perseguidos. Encada caso habrá que aplicar la expresión matemática apropiada:

- En el caso de la `superficie`, habrá que calcular la longitud de la base y la altura del rectángulo a partir de las coordenadas de las esquinas inferior izquierda (`x1, y1`) y superior derecha (`x2, y2`) de la figura. La base sería la diferencia entre `x2` y `x1`, y la altura la diferencia entre `y2` e `y1`. A continuación tan solo tendrías que utilizar la consabida fórmula de "base por altura", es decir, una multiplicación.
- En el caso del `perímetro` habrá también que calcular la longitud de la base y de la altura del rectángulo y a continuación sumar dos veces la longitud de la base y dos veces la longitud de la altura.

En ambos casos el resultado final tendrá que ser devuelto a través de la sentencia `return`. También es aconsejable en ambos casos la utilización de variables locales para almacenar los cálculos intermedios (como la base o la altura).

```

1 public double calcularSuperficie () {
2 double área, base, altura; // Variables locales
3 // Cálculo de la base
4 base = x2-x1;
5 // Cálculo de la altura
6 altura = y2-y1;
7 // Cálculo del área
8 área = base * altura;
9 // Devolución del valor de retorno
10 return área;
11 }
12
13 public double calcularPerímetro () {
14 double perímetro, base, altura; // Variables locales
15 // Cálculo de la base
16 base = x2-x1;
17 // Cálculo de la altura
18 altura = y2-y1;
19 // Cálculo del perímetro
20 perímetro = 2*base + 2*altura;
21 // Devolución del valor de retorno
22 return perímetro;
23 }
```

En el caso del método `desplazar()`, se trata de modificar:

- Los contenidos de los atributos `x1` y `x2` sumándoles el parámetro `X`,
- Los contenidos de los atributos `y1` e `y2` sumándoles el parámetro `Y`.

```

1 public void desplazar (double X, double Y) {
2 // Desplazamiento en el eje X
3 x1 = x1 + X;
4 x2 = x2 + X;
5 // Desplazamiento en el eje Y
6 y1 = y1 + Y;
7 y2 = y2 + Y;
8 }
```

En este caso no se devuelve ningún valor (tipo devuelto vacío: `void`).

Por último, el método `obtenerNumRectangulos` simplemente debe devolver el valor del atributo `numRectangulos`. En este caso es razonable plantearse que este método podría ser más bien un método de clase (estático) más que un método de objeto, pues en realidad es una característica de la clase más que algún objeto en particular. Para ello tan solo tendrías que utilizar el modificador de acceso `static`:

```

1 public static int obtenerNumRectangulos () {
2 return numRectangulos;
3 }
```

#### 16.13.4. Declaración de un objeto

Utilizando la clase `Rectangulo` implementada en ejercicios anteriores, indica como declararías tres objetos (variables) de esa clase llamados `r1`, `r2`, `r3`.

**Respuesta** Se trata simplemente de realizar una declaración de esas tres variables:

```

1 Rectangulo r1;
2 Rectangulo r2;
3 Rectangulo r3;
```

También podrías haber declarado los tres objetos en la misma sentencia de declaración:

```
1 Rectangulo r1, r2, r3;
```

#### 16.13.5. Creación de un objeto

Ampliar el ejercicio anterior instanciando los objetos `r1`, `r2`, `r3` mediante el constructor por defecto.

**Respuesta** Habría que añadir simplemente una sentencia de creación o instanciación (llamada al constructor mediante el operador `new`) por cada objeto que se deseé crear:

```

1 Rectangulo r1, r2, r3;
2 r1= new Rectangulo();
3 r2= new Rectangulo();
4 r3= new Rectangulo();

```

### 16.13.6. Manipulación de un objeto

Utilizar el ejemplo de los rectángulos para crear un rectángulo `r1`, asignarle los valores `x1 = 0, y1 = 0, x2 = 10, y2 = 10`, calcular su área y su perímetro y mostrarlos en pantalla.

**Respuesta** Se trata de declarar e instanciar el objeto `r1`, llenar sus atributos de ubicación (coordenadas de las esquinas), e invocar a los métodos `calcularSuperficie()` y `calcularPerimetro()` utilizando el operador punto (`.`).

Por ejemplo:

```

1 Rectangulo r1= new Rectangulo ();
2 r1.x1= 0;
3 r1.y1= 0;
4 r1.x2= 10;
5 r1.y2= 10;
6 area= r1.calcularSuperficie ();
7 perimetro= r1.calcularPerimetro ();

```

Por último faltaría mostrar en pantalla la información calculada, podemos añadir esta y más pruebas en un método `main` de la propia clase, o en el `main` de otra clase del mismo paquete, como prefieras.

### 16.13.7. Utilización de constructores

Vamos a ampliar el ejemplo anterior creando una clase `RectanguloV2`, ampliando sus funcionalidades añadiéndole tres constructores:

1. **Un constructor sin parámetros** (para sustituir al constructor por defecto) que haga que los valores iniciales de las esquinas del rectángulo sean (0,0) y (1,1);
2. **Un constructor con cuatro parámetros**, `x1, y1, x2, y2`, que rellene los valores iniciales de los atributos del rectángulo con los valores proporcionados a través de los parámetros.
3. **Un constructor con dos parámetros**, base y altura, que cree un rectángulo donde el vértice inferior izquierdo esté ubicado en la posición (0,0) y que tenga una base y una altura tal y como indican los dos parámetros proporcionados.

**Respuesta** En el caso del primer constructor lo único que hay que hacer es "rellenar" los atributos `x1, y1, x2, y2` con los valores 0, 0, 1, 1:

```

1 public RectanguloV2 (){
2 x1= 0.0;
3 y1= 0.0;
4 x2= 1.0;
5 y2= 1.0;
6 }

```

Para el segundo constructor es suficiente con asignar a los atributos `x1, y1, x2, y2` los valores de los parámetros `x1, y1, x2, y2`. Tan solo hay que tener en cuenta que al tener los mismos nombres los parámetros del método que los atributos de la clase, estos últimos son ocultados por los primeros y para poder tener acceso a ellos tendrás que utilizar el operador de autorreferencia `this`:

```

1 public RectanguloV2 (double x1, double y1, double x2, double y2){
2 this.x1= x1;
3 this.y1= y1;
4 this.x2= x2;
5 this.y2= y2;
6 }

```

En el caso del tercer constructor tendrás que inicializar el vértice (`x1, y1`) a (0,0) y el vértice (`x2, y2`) a (0 + base, 0 + altura), es decir a (base, altura):

```

1 public RectanguloV2 (double base, double altura) {
2 this.x1= 0.0;
3 this.y1= 0.0;
4 this.x2= base;
5 this.y2= altura;
6 }

```

Queda propuesto como ejercicio de ampliación la modificación de la clase `Rectangulo`, y ampliar el método `main` para usar las nuevas funcionalidades.

#### 16.13.8. Referencia `this`

Añadir un método `obtenerNombrev2` de la clase `Rectangulov2` de ejercicios anteriores utilizando la referencia `this`.

**Respuesta:** Si utilizamos la referencia `this` en este método, entonces podremos utilizar como identificador del parámetro el mismo identificador que tiene el atributo (aunque no tiene porqué hacerse si no se desea):

```
1 public void establecerNombrev2 (String nombre) {
2 this.nombre = nombre;
3 }
```

#### 16.13.9. Constructores de copia

Añadir un constructor de copia al ejercicio de la clase `Rectangulov2` usando referencias `this`.

**Respuesta** Se trata de añadir un nuevo constructor además de los tres que ya habíamos creado:

```
1 // Constructor copia
2 public Rectangulov2 (Rectangulov2 r) {
3 this.x1= r.x1;
4 this.y1= r.y1;
5 this.x2= r.x2;
6 this.y2= r.y2;
7 }
```

Para usar este constructor basta con haber creado anteriormente otro `Rectangulo` para utilizarlo como base de la copia. Por ejemplo:

```
1 Rectangulo r1, r2;
2 r1= new Rectangulo (0,0,2,2);
3 r2= new Rectangulo (r1);
```

#### 16.13.10. Ocultación de métodos

Vamos a intentar implementar una clase `DNI` que incluya todo lo que has visto hasta ahora. Se desea crear una clase que represente un DNI español y que tenga las siguientes características:

- La clase almacenará el número de `DNI` en un `int`, sin guardar la letra, pues se puede calcular a partir del número. Este atributo será privado a la clase. Formato del atributo: `private int numDNI`.
- Para acceder al `DNI` se dispondrá de dos métodos obtener (`get`), uno que proporcionará el número de `DNI` (sólo las cifras numéricas) y otro que devolverá el `NIF` completo (incluida la letra).

El formato del método será:

- `java public int obtenerDNI()`
- `java public String obtenerNIF()`
- Para modificar el `DNI` se dispondrá de dos métodos establecer (`set`), que permitirán modificar el `DNI`. Uno en el que habrá que proporcionar el `NIF` completo (número y letra). Y otro en el que únicamente será necesario proporcionar el `DNI` (las siete u ocho cifras). Si el `DNI/NIF` es incorrecto se debería lanzar algún tipo de excepción. El formato de los métodos (sobre cargados) será:
  - `java public void establecer (String nif) throws ...`
  - `java public void establecer (int dni) throws ...`
- La clase dispondrá de algunos métodos internos privados para calcular la letra de un número de `DNI` cualquiera, para comprobar si un `DNI` con su letra es válido, para extraer la letra de un `NIF`, etc. Aquellos métodos que no utilicen ninguna variable de objeto podrían declararse como estáticos (pertenecientes a la clase). Formato de los métodos:
  - `java private static char calcularLetraNIF (int dni)`
  - `java private boolean validarNIF (String nif)`
  - `java private static char extraerLetraNIF (String nif)`
  - `java private static int extraerNumeroNIF (String nif)`

Para calcular la letra **NIF** correspondiente a un número de **DNI** puedes consultar el artículo sobre el **NIF** de la [Wikipedia](#)

**Respuesta:**

Inténtalo por tu cuenta y cuando te quedes atascado tienes la solución en el apartado [Clase DNI](#)

## 16.14. Ejemplos UD05

### 16.14.1. Clase Rectangulo

```

1 package UD05;
2
3 class Rectangulo {
4
5 // Atributos de clase
6 private static int numRectangulos; // Número total de rectángulos creados
7 public static final String NOMBREFIGURA = "Rectángulo"; // Nombre de la clase
8 public static final double PI = 3.1416; // Constante PI
9
10 // Atributos de objeto
11 private String nombre; // Nombre del rectángulo
12 public double x1, y1; // Vértice inferior izquierdo
13 public double x2, y2; // Vértice superior derecho
14
15 // Método obtenerNombre
16 public String obtenerNombre() {
17 return nombre;
18 }
19
20 // Método establecerNombre
21 public void establecerNombre(String nom) {
22 nombre = nom;
23 }
24
25 // Método CalcularSuperficie
26 public double CalcularSuperficie() {
27 double area, base, altura;
28 // Cálculo de la base
29 base = x2 - x1;
30 // Cálculo de la altura
31 altura = y2 - y1;
32 // Cálculo del área
33 area = base * altura;
34 // Devolución del valor de retorno
35 return area;
36 }
37
38 // Método CalcularPerimetro
39 public double CalcularPerimetro() {
40 double perimetro, base, altura;
41 // Cálculo de la base
42 base = x2 - x1;
43 // Cálculo de la altura
44 altura = y2 - y1;
45 // Cálculo del perímetro
46 perimetro = 2 * base + 2 * altura;
47 // Devolución del valor de retorno
48 return perimetro;
49 }
50
51 // Método desplazar
52 public void desplazar(double X, double Y) {
53 // Desplazamiento en el eje X
54 x1 = x1 + X;
55 x2 = x2 + X;
56 // Desplazamiento en el eje Y
57 y1 = y1 + Y;
58 y2 = y2 + Y;
59 }
60
61 // Método obtenerNumRectangulos
62 public static int obtenerNumRectangulos() {
63 return numRectangulos;
64 }
65
66 public static void main(String[] args) {
67 Rectangulo r1, r2;
68 r1 = new Rectangulo();
69 r2 = new Rectangulo();
70 r1.x1 = 0;
71 r1.y1 = 0;
72 r1.x2 = 10;
73 r1.y2 = 10;
74 r1.establecerNombre("rectangulo1");
75 System.out.printf("PRUEBA DE USO DE LA CLASE " + Rectangulo.NOMBREFIGURA + "\n");
76 System.out.printf("-----\n");
77 System.out.printf("r1.x1: %.2f\nr1.y1: %.2f\n", r1.x1, r1.y1);
78 System.out.printf("r1.x2: %.2f\nr1.y2: %.2f\n", r1.x2, r1.y2);
79 System.out.printf("Perímetro: %.2f\nSuperficie: %.2f\n",
80 r1.CalcularPerimetro(), r1.CalcularSuperficie());
81 System.out.printf("Desplazamos X=3, Y=3\n");
82 r1.desplazar(-3, 3);
83 System.out.printf("r1.x1: %.2f\nr1.y1: %.2f\n", r1.x1, r1.y1);
84 System.out.printf("r1.x2: %.2f\nr1.y2: %.2f\n", r1.x2, r1.y2);
85 }
86}

```

### 16.14.2. Clase Rectangulov2

```

1 package UD05;
2
3 class Rectangulov2 {
4
5 // Atributos de clase
6 private static int numRectangulos; // Número total de rectángulos creados
7 public static final String NOMBRFIGURA = "Rectángulov2"; // Nombre de la clase
8 public static final double PI = 3.1416; // Constante PI
9
10 // Atributos de objeto
11 private String nombre; // Nombre del rectángulo
12 public double x1, y1; // Vértice inferior izquierdo
13 public double x2, y2; // Vértice superior derecho
14
15 // Método obtenerNombre
16 public String obtenerNombre() {
17 return nombre;
18 }
19
20 // Método establecerNombre
21 public void establecerNombre(String nom) {
22 nombre = nom;
23 }
24
25 // Método CalcularSuperficie
26 public double CalcularSuperficie() {
27 double area, base, altura;
28 // Cálculo de la base
29 base = x2 - x1;
30 // Cálculo de la altura
31 altura = y2 - y1;
32 // Cálculo del área
33 area = base * altura;
34 // Devolución del valor de retorno
35 return area;
36 }
37
38 // Método CalcularPerímetro
39 public double CalcularPerímetro() {
40 double perímetro, base, altura;
41 // Cálculo de la base
42 base = x2 - x1;
43 // Cálculo de la altura
44 altura = y2 - y1;
45 // Cálculo del perímetro
46 perímetro = 2 * base + 2 * altura;
47 // Devolución del valor de retorno
48 return perímetro;
49 }
50
51 // Método desplazar
52 public void desplazar(double X, double Y) {
53 // Desplazamiento en el eje X
54 x1 = x1 + X;
55 x2 = x2 + X;
56 // Desplazamiento en el eje Y
57 y1 = y1 + Y;
58 y2 = y2 + Y;
59 }
60
61 // Método obtenerNumRectangulos
62 public static int obtenerNumRectangulos() {
63 return numRectangulos;
64 }
65
66 // Constructor por defecto
67 public Rectangulov2() {
68 x1 = 0.0;
69 y1 = 0.0;
70 x2 = 1.0;
71 y2 = 1.0;
72 numRectangulos++;
73 }
74
75 // constructor con los 4 vértices
76 public Rectangulov2(double x1, double y1, double x2, double y2) {
77 this.x1 = x1;
78 this.y1 = y1;
79 this.x2 = x2;
80 this.y2 = y2;
81 numRectangulos++;
82 }

```

```
1 //constructor con base y altura
2 public Rectangulov2(double base, double altura) {
3 this.x1 = 0.0;
4 this.y1 = 0.0;
5 this.x2 = base;
6 this.y2 = altura;
7 numRectangulos++;
8 }
9
10 //referencia this
11 public void establecerNombrer2(String nombre) {
12 this.nombre = nombre;
13 }
14
15 // Constructor copia
16 public Rectangulov2(Rectangulov2 r) {
17 this.nombre=r.nombre;//supongo que también quiero copiar el nombre
18 this.x1 = r.x1;
19 this.y1 = r.y1;
20 this.x2 = r.x2;
21 this.y2 = r.y2;
22 numRectangulos++;
23 }
24
25 public static void main(String[] args) {
26 Rectangulov2 r1;
27 Rectangulov2 r2;
28 Rectangulov2 r3;
29 r1 = new Rectangulov2();
30 r2 = new Rectangulov2(4, 4, 8, 8);
31 r3 = new Rectangulov2(5, 5);
32 r1.establecerNombrer2("defecto");
33 r2.establecerNombrer2("4 vertices");
34 r3.establecerNombrer2("base y altura");
35
36 System.out.printf("PRUEBA DE USO DE LA CLASE " + Rectangulov2.NOMBREFIGURA + "\n");
37 System.out.printf("-----\n");
38 System.out.printf("r1.x1: %4.2f\nr1.y1: %4.2f\n", r1.x1, r1.y1);
39 System.out.printf("r1.x2: %4.2f\nr1.y2: %4.2f\n", r1.x2, r1.y2);
40
41 //Usamos el constructor de copia para realizar una copia del rectángulo
42 Rectangulov2 r4 = new Rectangulov2(r1);
43 System.out.println("r4 es una copia de r1");
44
45 System.out.printf("r4.x1: %4.2f\nr4.y1: %4.2f\n", r4.x1, r4.y1);
46 System.out.printf("r4.x2: %4.2f\nr4.y2: %4.2f\n", r4.x2, r4.y2);
47 }
48 }
```

**16.14.3. Clase DNI**

```

1 public class DNI {
2
3 // Atributos estáticos
4 // Cadena con las letras posibles del DNI ordenados para el cálculo de DNI
5 private static final String LETRAS_DNI = "TRWAGMYFPDXBNJZSQVHLCKE";
6
7 // Atributos de objeto
8 private int numDNI;
9
10 // Métodos
11 public String obtenerNIF() {
12 // Variables locales
13 String cadenaNIF;
14 // NIF con letra para devolver
15 char letraNIF;
16 // Letra del número de NIF calculado
17 // Cálculo de la letra del NIF
18 letraNIF = calcularLetraNIF(numDNI);
19 // Construcción de la cadena del DNI: número + letra
20 cadenaNIF = Integer.toString(numDNI) + String.valueOf(letraNIF);
21 // Devolución del resultado
22 return cadenaNIF;
23 }
24
25 public int obtenerDNI() {
26 return numDNI;
27 }
28
29 public void establecer(String nif) throws Exception {
30 if (DNI.validarNIF(nif)) { // Valor válido: lo almacenamos
31 this.numDNI = DNI.extraerNumeroNIF(nif);
32 } else { // Valor inválido: lanzamos una excepción
33 throw new Exception("NIF inválido: " + nif);
34 }
35 }
36
37 public void establecer(int dni) throws Exception {
38 // Comprobación de rangos
39 if (dni > 999999 && dni < 99999999) {
40 this.numDNI = dni; // Valor válido: lo almacenamos
41 } else { // Valor inválido: lanzamos una excepción
42 throw new Exception("DNI inválido: " + String.valueOf(dni));
43 }
44 }
45
46 private static char calcularLetraNIF(int dni) {
47 char letra;
48 // Cálculo de la letra NIF
49 letra = LETRAS_DNI.charAt(dni % 23);
50 // Devolución de la letra NIF
51 return letra;
52 }
53
54 private static char extraerLetraNIF(String nif) {
55 char letra = nif.charAt(nif.length() - 1);
56 return letra;
57 }
58
59 private static int extraerNumeroNIF(String nif) {
60 int numero = Integer.parseInt(nif.substring(0, nif.length() - 1));
61 return numero;
62 }
63
64 private static boolean validarNIF(String nif) {
65 boolean valido = true;
66 // Suponemos el NIF válido mientras no se encuentre algún fallo
67 char letra_calculada;
68 char letra_leida;
69 int dni_leido;
70 if (nif == null) { // El parámetro debe ser un objeto no vacío
71 valido = false;
72 } else if (nif.length() < 8 || nif.length() > 9) {
73 // La cadena debe estar entre 8(7+1) y 9(8+1) caracteres
74 valido = false;
75 } else {
76 letra_leida = DNI.extraerLetraNIF(nif);
77 // Extraemos la letra de NIF (letra)
78 dni_leido = DNI.extraerNumeroNIF(nif); // Extraemos el número de DNI (int)
79 letra_calculada = DNI.calcularLetraNIF(dni_leido);
80 // Calculamos la letra de NIF a partir del número extraído
81 if (letra_leida == letra_calculada) {
82 // Comparamos la letra extraída con la calculada
83 // Todas las comprobaciones han resultado válidas. El NIF es válido.
84 valido = true;
85 } else {
86 valido = false;
87 }
88 }
89 return valido;
90 }
91 }

```

#### 16.14.4. Casting

```

1 package UD05;
2
3 public class Casting {
4
5 public static void main(String[] args) {
6 // Casting Implicito
7 Persona encargadoCarniceria = new Encargado("Rosa Ramos", 1200,
8 "carniceria");
9
10 // No tenemos disponibles los métodos de la clase Encargado:
11 //EncargadaCarniceria.setSueldoBase(1200);
12 //EncargadaCarniceria.setSeccion("Carniceria");
13 //Pero al imprimir se imprime con el método más específico (luego lo vemos)
14 System.out.println(encargadoCarniceria);
15
16 // Casting Explicito
17 Encargado miEncargado = (Encargado) encargadoCarniceria;
18 //Tenemos disponibles los métodos de la clase Encargado:
19 miEncargado.setSueldoBase(1200);
20 miEncargado.setSeccion("Carniceria");
21 //Al imprimir se imprime con el método más específico de nuevo.
22 System.out.println(miEncargado);
23 }
24 }
```

##### 16.14.4.1. Persona

```

1 package UD05;
2
3 // Clase Persona que solo dispone de nombre
4 public class Persona {
5
6 String nombre;
7
8 public Persona(String nombre) {
9 this.nombre = nombre;
10 }
11
12 public void setNombre(String nom) {
13 nombre = nom;
14 }
15
16 public String getNombre() {
17 return nombre;
18 }
19
20 @Override
21 public String toString() {
22 return "Nombre: " + nombre;
23 }
24 }
```

##### 16.14.4.2. Empleado

```

1 package UD05;
2
3 // Clase Empleado que hereda de Persona y añade atributo sueldoBase
4 public class Empleado extends Persona {
5
6 double sueldoBase;
7
8 public Empleado(String nombre, double sueldoBase) {
9 super(nombre);
10 this.sueldoBase = sueldoBase;
11 }
12
13 public double getSueldo() {
14 return sueldoBase;
15 }
16
17 public void setSueldoBase(double sueldoBase) {
18 this.sueldoBase = sueldoBase;
19 }
20
21 @Override
22 public String toString() {
23 return super.toString() + "\nSueldo Base: " + sueldoBase;
24 }
25 }
```

#### 16.14.5. Encargado

```

1 package UD05;
2
3 // Clase Encargado que hereda de Empleado y añade atributo seccion
4 public class Encargado extends Empleado {
5
6 String seccion;
7
8 public Encargado(String nombre, double sueldoBase, String seccion) {
9 super(nombre, sueldoBase);
10 this.seccion = seccion;
11 }
12
13 public String getSeccion() {
14 return seccion;
15 }
16
17 public void setSeccion(String seccion) {
18 this.seccion = seccion;
19 }
20
21 @Override
22 public String toString() {
23 return super.toString() + "\nSección:" + seccion ;
24 }
25 }
```

#### 16.14.6. ClasesAnidadas

```

1 package UD05;
2
3 class Pc {
4
5 double precio;
6
7 public String toString() {
8 return "El precio del PC es " + this.precio;
9 }
10
11 class Monitor {
12
13 String marca;
14
15 public String toString() {
16 return "El monitor es de la marca " + this.marca;
17 }
18 }
19
20 class Cpu {
21
22 String marca;
23
24 public String toString() {
25 return "La CPU es de la marca " + this.marca;
26 }
27 }
28 }
29
30 public class ClasesAnidadas {
31
32 public static void main(String[] args) {
33 Pc miPc = new Pc();
34 Pc.Monitor miMonitor = miPc.new Monitor();
35 Pc.Cpu miCpu = miPc.new Cpu();
36 miPc.precio = 1250.75;
37 miMonitor.marca = "Asus";
38 miCpu.marca = "Acer";
39 System.out.println(miPc); //El precio del PC es 1250.75
40 System.out.println(miMonitor); //El monitor es de la marca Asus
41 System.out.println(miCpu); //La CPU es de la marca Acer
42 }
43 }
```

### 16.15. Píldoras informáticas relacionadas

⌚29 de diciembre de 2025

## 17. 6.2 Anexo Wrappers y Fechas

### 17.1. Wrappers (Envoltorios)

Los wrappers permiten "envolver" datos primitivos en objetos, también se llaman clases contenedoras. La diferencia entre un tipo primitivo y un wrapper es que este último es una clase y por tanto, cuando trabajamos con wrappers estamos trabajando con objetos.



Como son objetos debemos tener cuidado en el paso como parámetro en métodos ya que en el wrapper se realiza por referencia.

Una de las principales ventajas del uso de wrappers son la facilidad de conversión entre tipos primitivos y cadenas.

Hay una clase contenedora por cada uno de los tipos primitivos de Java. Los datos primitivos se escriben en minúsculas y los wrappers se escriben con la primera letra en mayúsculas.

| Tipo primitivo | Wrapper asociado |
|----------------|------------------|
| byte           | Byte             |
| short          | Short            |
| int            | Integer          |
| long           | Long             |
| float          | Float            |
| double         | Double           |
| char           | Char             |
| boolean        | Boolean          |

Cada clase wrapper tiene dos constructores, uno se le pasa por parámetro el dato de tipo primitivo y otro se le pasa un `String`.

Para wrapper `Integer`:

```
1 Integer(int)
2 Integer(String)
```

Ejemplo:

```
1 Integer i1 = new Integer(42);
2 Integer i2 = new Integer ("42");
3 Float f1 = new Float(3.14f);
4 Float f2 = new Float ("3.14f");
```

Antiguamente, una vez asignado un valor a un objeto o wrapper `Integer`, este no podía cambiarse. Actualmente e internamente se apoyan en variables y wrappers internos para poder variar el valor de un wrapper.

Ejemplo:

```
1 Integer y = new Integer(567); //Crea el objeto
2 y++; //Lo desenvuelve, incrementa y lo vuelve a envolver
3 System.out.println("Valor: " + y); //Imprime el valor del objeto y
```

Los wrapper disponen de una serie de métodos que permiten realizar funciones de conversión de datos. Por ejemplo, el wrapper `Integer` dispone de los siguientes métodos:

| Método                       | Descripción   |
|------------------------------|---------------|
| <code>Integer(int)</code>    | Constructores |
| <code>Integer(String)</code> |               |

| Método                                                                                                                                                                                               | Descripción                                  |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------|
| <code>byteValue()</code><br><code>shortValue()</code><br><code>intValue()</code><br><code>longValue()</code><br><code>doubleValue()</code><br><code>floatValue()</code>                              | Funciones de conversión con datos primitivos |
| <code>Integer decode(String)</code><br><code>Integer parseInt(String)</code><br><code>Integer parseInt(String, int)</code><br><code>Integer valueOf(String)</code><br><code>String toString()</code> | Conversión a String                          |
| <code>String toBinaryString(int)</code><br><code>String toHexString(int)</code><br><code>String toOctalString(int)</code>                                                                            | Conversión a otros sistemas de numeración    |
| <code>MAX_VALUE</code> , <code>MIN_VALUE</code> , <code>TYPE</code>                                                                                                                                  | Constantes                                   |

### 17.1.1. Métodos `valueOf()`

El método `valueOf()` permite crear objetos wrapper y se le pasa un parámetro `String` y opcionalmente otro parámetro que indica la base en la que será representado el primer parámetro.

Ejemplo:

```

1 // Convierte el 101011 (base 2) a 43 y le asigna el valor al objeto Integer i3
2 Integer i3 = Integer.valueOf("101011", 2);
3 System.out.println(i3);
4
5 // Asigna 3.14 al objeto Float f3
6 Float f3 = Float.valueOf("3.14f");
7 System.out.println(f3);

```

### 17.1.2. Métodos `xxxValue()`.

Los métodos `xxxValue()` permiten convertir un wrapper en un dato de tipo primitivo y no necesitan argumentos.

Ejemplo:

```

1 Integer i4 = 120; // Crea un nuevo objeto wrapper
2 byte b = i4.byteValue(); // Convierte el valor de i4 a un primitivo byte
3 short s1 = i4.shortValue(); // Otro de los métodos de Integer
4 double d = i4.doubleValue(); // Otro de los métodos xxxValue de Integer
5 System.out.println(s1); // Muestra 120 como resultado
6
7 Float f4 = 3.14f; // Crea un nuevo objeto wrapper
8 short s2 = f4.shortValue(); // Convierte el valor de f4 en un primitivo short
9 System.out.println(s2); // El resultado es 3 (truncado, no redondeado)

```

### 17.1.3. Métodos `parseXXXX()`

Los métodos `parseXXXX()` permiten convertir un wrapper en un dato de tipo primitivo y le pasamos como parámetro el `String` con el valor que deseamos convertir y opcionalmente la base a la que convertiremos el valor (2, 8, 10 o 16).

Ejemplo:

```

1 double d4 = Double.parseDouble("3.14"); // Convierte un String a primitivo
2 System.out.println("d4 = " + d4); // El resultado será d4 = 3.14
3 long l2 = Long.parseLong("101010", 2); // un String binario a primitivo
4 System.out.println("l2 = " + l2); // El resultado es l2 = 42

```

### 17.1.4. Métodos `toString()`

El método `toString()` permite retornar un `String` con el valor primitivo que se encuentra en el objeto contenedor. Se le pasa un parámetro que es el wrapper y opcionalmente para `Integer` y `Long` un parámetro con la base a la que convertiremos el valor (2, 8, 10 o 16).

Ejemplo:

```

1 Double d1 = new Double("3.14");
2 System.out.println("d1 = " + d1.toString()); // El resultado es d = 3.14
3 String d2 = Double.toString(3.14); // d2 = "3.14"
4 System.out.println("d2 = " + d2); // El resultado es d = 3.14
5 String s3 = "hex = " + Long.toHexString(254, 16); // s3 = "hex = fe"
6 System.out.println("s3 = " + s3); // El resultado es s3 = hex = fe

```

### 17.1.5. Métodos `toXXXXString()` (Binario, Hexadecimal y Octal)

Los métodos `toXXXXString()` permiten a las clases contenedoras `Integer` y `Long` convertir números en base 10 a otras bases, retornando un `String` con el valor primitivo que se encuentra en el objeto contenedor.

Ejemplo:

```

1 String s4 = Integer.toHexString(254); // Convierte 254 a hex
2 System.out.println("254 es " + s4); // Resultado: "254 es fe"
3 String s5 = Long.toOctalString(254); // Convierte 254 a octal
4 System.out.println("254(oct) = " + s5); // Resultado: "254(oct) = 376"

```

#### i Resumen

Para resumir, los métodos esenciales para las conversiones son:

- `primitive xxxValue()` - Para convertir de `Wrapper` a `primitive`
- `primitive parseXXX(String)` - Para convertir un `String` en `primitive`
- `Wrapper valueOf(String)` - Para convertir `String` en `Wrapper`

## 17.2. Clase Date

La clase `Date` es una utilidad contenida en el paquete `java.util` y permiten trabajar con fechas y horas. La fechas y hora se almacenan en un entero de tipo `Long` que almacena los milisegundos transcurridos desde el 1 de Enero de 1970 que se obtienen con `getTime()`. (Importamos `java.util.Date`).

Ejemplo:

```

1 Date fecha = new Date(2021, 9, 19);
2 System.out.println(fecha); //Mon Sep 19 00:00:00 CEST 2021
3 System.out.println(fecha.getTime()); //61590146400000

```

### 17.2.1. Clase GregorianCalendar

Para utilizar fechas y horas se utiliza la clase `GregorianCalendar` que dispone de variables enteras como: `DAY_OF_WEEK`, `DAY_OF_MONTH`, `YEAR`, `MONTH`, `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`, `WEEK_OF_MONTH`, `WEEK_OF_YEAR`, ... (Importamos Clase `java.util.Calendar` y `java.util.GregorianCalendar`)

Ejemplo 1:

```

1 Calendar calendar = new GregorianCalendar(2021, 9, 19);
2 System.out.println(calendar.getTime()); //Sun Sep 19 00:00:00 CEST 2021

```

Ejemplo 2:

```

1 Date d = new Date();
2 GregorianCalendar c = new GregorianCalendar();
3 System.out.println("Fecha: "+d); //Fecha: Thu Aug 19 20:06:14 CEST 2021
4 System.out.println("Info: "+c); //Info:
5 //java.util.GregorianCalendar[time=1629396374723,areFieldsSet=true
6 //,areAllFieldsSet=true
7 //,lenient=true,zone=sun.util.calendar.ZoneInfo[id="Europe/Madrid",offset=3600000
8 //,dstSavings=3600000,useDaylight=true,transitions=163
9 //,lastRule=jav.util.SimpleTimeZone[id=Europe/Madrid,offset=3600000
10 //,dstSavings=3600000,useDaylight=true,startYear=0,startMode=2,startMonth=2
11 //,startDay=-1,startDayOfWeek=1,startTime=3600000,startTimeMod2.1e=2,endMode=2
12 //,endMonth=9,endDay=-1,endDayOfWeek=1,endTime=3600000,endTimeMode=2]
13 //,firstDayOfWeek=2,minimalDaysInFirstWeek=4,ERA=1,YEAR=2021,MONTH=7,WEEK_OF_YEAR=33
14 //,WEEK_OF_MONTH=3,DAY_OF_MONTH=19,DAY_OF_YEAR=231,DAY_OF_WEEK=5
15 //,DAY_OF_WEEK_IN_MONTH=3,AM_PM=1,HOUR=8,HOUR_OF_DAY=20,MINUTE=6,SECOND=14
16 //,MILLISCOND=723,ZONE_OFFSET=3600000,DST_OFFSET=3600000]
17 c.setTime(d);
18 System.out.print(c.get(Calendar.DAY_OF_MONTH));//19
19 System.out.print("/");
20 System.out.print(c.get(Calendar.MONTH)+1); //9
21 System.out.print("/");
22 System.out.println(c.get(Calendar.YEAR)); //2022

```

## 17.2.2. Paquete java.time

El paquete `java.time` dispone de las clases `LocalDate`, `LocalTime`, `LocalDateTime`, `Duration` y `Period` para trabajar con fechas y horas.

Estas clases no tienen constructores públicos, y por tanto, no se puede usar `new` para crear objetos de estas clases. Necesitas usar sus métodos `static` para instanciarlas.

No es válido llamar directamente al constructor usando `new`, ya que no tienen un constructor público.

Ejemplo:

```
1 LocalDate d = new LocalDate(); //NO compila
```

### 17.2.2.1. LocalDate

`LocalDate` representa una fecha determinada. Haciendo uso del método `of()`, esta clase puede crear un `LocalDate` teniendo en cuenta el año, mes y día. Finalmente, para capturar el `LocalDate` actual se puede usar el método `now()`:

Ejemplo:

```

1 LocalDate date = LocalDate.of(1989, 11, 11); //1989-11-11
2 System.out.println(date.getYear()); //1989
3 System.out.println(date.getMonth()); //NOVEMBER
4 System.out.println(date.getDayOfMonth()); //11
5 date = LocalDate.now();
6 System.out.println(date); //2021-08-19

```

### 17.2.2.2. LocalTime

`LocalTime`, representa un tiempo determinado. Haciendo uso del método `of()`, esta clase puede crear un `LocalTime` teniendo en cuenta la hora, minuto, segundo y nanosegundo. Finalmente, para capturar el `LocalTime` actual se puede usar el método `now()`.

```

1 LocalTime time = LocalTime.of(5, 30, 45, 35); //05:30:45:35
2 System.out.println(time.getHour()); //5
3 System.out.println(time.getMinute()); //30
4 System.out.println(time.getSecond()); //45
5 System.out.println(time.getNano()); //35
6 time = LocalTime.now();
7 System.out.println(time); //20:13:53.118044

```

### 17.2.2.3. LocalDateTime

`LocalDateTime`, es una clase compuesta, la cual combina las clases anteriormente mencionadas `LocalDate` y `LocalTime`. Podemos construir un `LocalDateTime` haciendo uso de todos los campos (año, mes, día, hora, minuto, segundo, nanosegundo).

Ejemplo:

```
1 LocalDateTime dateTime = LocalDateTime.of(1989, 11, 11, 5, 30, 45, 35);
```

También, se puede crear un objeto `LocalDateTime` basado en los tipos `LocalDate` y `LocalTime`, haciendo uso del método `of()` (`LocalDate date`, `LocalTime time`):

Ejemplo:

```
1 LocalDate date = LocalDate.of(1989, 11, 11);
2 LocalTime time = LocalTime.of(5, 30, 45, 35);
3 LocalDateTime dateTime = LocalDateTime.of(date, time);
4 LocalDateTime dateTime = LocalDateTime.now();
```

#### 17.2.2.4. Duration

`Duration`, hace referencia a la diferencia que existe entre dos objetos de tiempo. La duración denota la cantidad de tiempo en horas, minutos y segundos.

Ejemplo:

```
1 LocalTime localTime1 = LocalTime.of(12, 25);
2 LocalTime localTime2 = LocalTime.of(17, 35);
3 Duration duration1 = Duration.between(localTime1, localTime2);
4 System.out.println(duration1); //PT5H10M
5 System.out.println(duration1.toDays()); //0
6
7 LocalDateTime localDateTime1 = LocalDateTime.of(2016, Month.JULY, 18, 14, 13);
8 LocalDateTime localDateTime2 = LocalDateTime.of(2016, Month.JULY, 20, 12, 25);
9 Duration duration2 = Duration.between(localDateTime1, localDateTime2);
10 System.out.println(duration2); //PT46H12M
11 System.out.println(duration2.toDays()); //1
```

También, se puede crear `Duration` basado en los métodos `ofDays(long days)`, `ofHours(long hours)`, `ofMilis(long milis)`, `ofMinutes(long minutes)`, `ofNanos(long nanos)`, `ofSeconds(long seconds)`.

Ejemplo:

```
1 Duration duracion3 = Duration.ofDays(1);
2 System.out.println(duracion3); //PT24H
3 System.out.println(duracion3.toDays()); //1
```

#### 17.2.2.5. Period

`Period`, hace referencia a la diferencia que existe entre dos fechas. Esta clase denota la cantidad de tiempo en años, meses y días.

```
1 LocalDate localDate1 = LocalDate.of(2016, 7, 18);
2 LocalDate localDate2 = LocalDate.of(2016, 7, 20);
3 Period periodo1 = Period.between(localDate1, localDate2);
4 System.out.println(periodo1); //P2D
```

Se puede crear `Period` basado en el método `of(int years, int months, int days)`. En el siguiente ejemplo, se crea un período de 1 año 2 meses y 3 días:

```
1 Period periodo2 = Period.of(1, 2, 3);
2 System.out.println(periodo2); //P1Y2M3D
```

Se puede crear `Period` basado en los métodos `ofDays(int days)`, `ofMonths(int months)`, `ofWeeks(int weeks)`, `ofYears(int years)`.

Ejemplo:

```
1 Period periodo3 = Period.ofYears(1);
2 System.out.println(periodo3); //P1Y
```

#### 17.2.3. ChronoUnit

Permite devolver el tiempo transcurrido entre dos fechas en diferentes formatos (`DAYS`, `MONTHS`, `YEARS`, `HOURS`, `MINUTES`, `SECONDS`, ...). Debemos importar la clase `java.time.temporal.ChronoUnit`;

Ejemplo:

```
1 LocalDate fechaInicio = LocalDate.of(2016, 7, 18);
2 LocalDate fechaFin = LocalDate.of(2016, 7, 20);
3 // Calculamos el tiempo transcurrido entre las dos fechas
4 // con la clase ChronoUnit y la unidad temporal en la que
5 // queremos que nos lo devuelva, en este caso DAYS.
6 long tiempo = ChronoUnit.DAYS.between(fechaInicio, fechaFin);
7 System.out.println(tiempo); //2
```

#### 17.2.4. Introducir fecha como Cadena

Podemos introducir la fecha como una cadena con el formato que deseemos y posteriormente convertir a fecha con la sentencia `parse`. Debemos importar las clases `time` y `time.format`.

Ejemplo:

```
1 DateTimeFormatter formato = DateTimeFormatter.ofPattern("d/MM/u");
2 String fechaCadena = "16/08/2016";
3 LocalDate mifecha = LocalDate.parse(fechaCadena, formato);
4 System.out.println(formato.format(mifecha)); //16/08/2016
```



A partir de Java 8 `y` es para el año de la era (BC AD), y para el año debemos usar `u`

Más detalles sobre los formatos: <https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html>

#### 17.2.5. Manipulación

##### 1. Manipulando `LocalDate`

Haciendo uso de los métodos `withYear(int year)`, `withMonth(int month)`, `withDayOfMonth(int dayOfMonth)`, `with(TemporalField field, long newValue)` se puede modificar el `LocalDate`.

Ejemplo:

```
1 LocalDate date = LocalDate.of(2016, 7, 25);
2 LocalDate date1 = date.withYear(2017);
3 LocalDate date2 = date.withMonth(8);
4 LocalDate date3 = date.withDayOfMonth(27);
5 System.out.println(date); //2016-07-25
6 System.out.println(date1); //2017-07-25
7 System.out.println(date2); //2016-08-25
8 System.out.println(date3); //2016-07-27
```

##### 1. Manipulando `LocalTime`

Haciendo uso de los métodos `withHour(int hour)`, `withMinute(int minute)`, `withSecond(int second)`, `withNano(int nanoOfSecond)` se puede modificar el `LocalTime`.

Ejemplo:

```
1 LocalTime time = LocalTime.of(14, 30, 35);
2 LocalTime time1 = time.withHour(20);
3 LocalTime time2 = time.withMinute(25);
4 LocalTime time3 = time.withSecond(23);
5 LocalTime time4 = time.withNano(24);
6 System.out.println(time); //14:30:35
7 System.out.println(time1); //20:30:35
8 System.out.println(time2); //14:25:35
9 System.out.println(time3); //14:30:23
10 System.out.println(time4); //14:30:35.000000024
```

##### 1. Manipulando `LocalDateTime`

`LocalDateTime` provee los mismo métodos mencionados en las clases `LocalDate` y `LocalTime`.

Ejemplo:

```

1 LocalDateTime dateTime = LocalDateTime.of(2016, 7, 25, 22, 11, 30);
2 LocalDateTime dateTime1 = dateTime.withYear(2017);
3 LocalDateTime dateTime2 = dateTime.withMonth(8);
4 LocalDateTime dateTime3 = dateTime.withDayOfMonth(27);
5 LocalDateTime dateTime4 = dateTime.withHour(20);
6 LocalDateTime dateTime5 = dateTime.withMinute(25);
7 LocalDateTime dateTime6 = dateTime.withSecond(23);
8 LocalDateTime dateTime7 = dateTime.withNano(24);
9 System.out.println(dateTime); //2016-07-25T22:11:30
10 System.out.println(dateTime1); //2017-07-25T22:11:30
11 System.out.println(dateTime2); //2016-08-25T22:11:30
12 System.out.println(dateTime3); //2016-07-27T22:11:30
13 System.out.println(dateTime4); //2016-07-25T20:11:30
14 System.out.println(dateTime5); //2016-07-25T22:25:30
15 System.out.println(dateTime6); //2016-07-25T22:11:23
16 System.out.println(dateTime7); //2016-07-25T22:11:30.000000024

```

## 17.2.6. Operaciones

### 17.2.6.1. OPERACIONES CON `LocalDate`

Realizar operaciones como suma o resta de días, meses, años, etc es muy fácil con la nueva `Date API`. Los siguientes métodos `plus(long amountToAdd, TemporalUnit unit)`, `minus(long amountToSubtract, TemporalUnit unit)` proveen una manera general de realizar estas operaciones. (Debemos importar la clase `java.time.temporal.ChronoUnit` para poder utilizar las unidades: `ChronoUnit.YEARS`, `ChronoUnit.MONTHS`, `ChronoUnit.DAYS`).

Ejemplo:

```

1 LocalDate date = LocalDate.of(2016, 7, 18);
2 LocalDate datePlusOneDay = date.plus(1, ChronoUnit.DAYS);
3 LocalDate dateMinusOneDay = date.minus(1, ChronoUnit.DAYS);
4 System.out.println(date); // 2016-07-18
5 System.out.println(datePlusOneDay); // 2016-07-19
6 System.out.println(dateMinusOneDay); // 2016-07-17

```

También se puede hacer cálculos basados en un `Period`. En el siguiente ejemplo, se crea un `Period` de 1 día para poder realizar los cálculos.

Ejemplo:

```

1 LocalDate date = LocalDate.of(2016, 7, 18);
2 LocalDate datePlusOneDay = date.plus(Period.ofDays(1));
3 LocalDate dateMinusOneDay = date.minus(Period.ofDays(1));
4 System.out.println(date); // 2016-07-18
5 System.out.println(datePlusOneDay); // 2016-07-19
6 System.out.println(dateMinusOneDay); // 2016-07-17

```

Finalmente, haciendo uso de métodos explícitos como `plusDays(long daysToAdd)` y `minusDays(long daysToSubtract)` se puede indicar el valor a incrementar o reducir.

Ejemplo:

```

1 LocalDate date = LocalDate.of(2016, 7, 18);
2 LocalDate datePlusOneDay = date.plusDays(1);
3 LocalDate dateMinusOneDay = date.minusDays(1);
4 System.out.println(date); // 2016-07-18
5 System.out.println(datePlusOneDay); // 2016-07-19
6 System.out.println(dateMinusOneDay); // 2016-07-17

```

### 17.2.6.2. OPERACIONES CON `LocalTime`

La nueva `Date API` permite realizar operaciones como suma y resta de horas, minutos, segundos, etc. Al igual que `LocalDate`, los siguientes métodos `plus(long amountToAdd, TemporalUnit unit)`, `minus(long amountToSubtract, TemporalUnit unit)` proveen una manera general de realizar estas operaciones.

(Debemos importar la clase `java.time.temporal.ChronoUnit` para poder utilizar las unidades: `ChronoUnit.HOURS`, `ChronoUnit.MINUTES`, `ChronoUnit.SECONDS`, `ChronoUnit.NANOS`).

Ejemplo:

```

1 LocalTime time = LocalTime.of(15, 30);
2 LocalTime timePlusOneHour = time.plus(1, ChronoUnit.HOURS);
3 LocalTime timeMinusOneHour = time.minus(1, ChronoUnit.HOURS);
4 System.out.println(time); // 15:30
5 System.out.println(timePlusOneHour); // 16:30
6 System.out.println(timeMinusOneHour); // 14:30

```

También se puede hacer cálculos basados en un `Duration`. En el siguiente ejemplo, se crea un `Duration` de 1 hora para poder realizar los cálculos.

```

1 LocalTime time = LocalTime.of(15, 30);
2 LocalTime timePlusOneHour = time.plus(Duration.ofHours(1));
3 LocalTime timeMinusOneHour = time.minus(Duration.ofHours(1));
4 System.out.println(time); // 15:30
5 System.out.println(timePlusOneHour); // 16:30
6 System.out.println(timeMinusOneHour); // 14:30

```

Finalmente, haciendo uso de métodos explícitos como `plusHours(long hoursToAdd)` y `minusHours(long hoursToSubtract)` se puede indicar el valor a incrementar o reducir.

Ejemplo:

```

1 LocalTime time = LocalTime.of(15, 30);
2 LocalTime timePlusOneHour = time.plusHours(1);
3 LocalTime timeMinusOneHour = time.minusHours(1);
4 System.out.println(time); // 15:30
5 System.out.println(timePlusOneHour); // 16:30
6 System.out.println(timeMinusOneHour); // 14:30

```

#### 17.2.6.3. OPERACIONES CON `LocalDateTime`

`LocalDateTime`, al ser una clase compuesta por `LocalDate` y `LocalTime` ofrece los mismos métodos para realizar operaciones.

(Debemos importar la clase `java.time.temporal.ChronoUnit` para poder utilizar las unidades: `ChronoUnit.YEARS`, `ChronoUnit.MONTHS`, `ChronoUnit.DAYS`, `ChronoUnit.HOURS`, `ChronoUnit.MINUTES`, `ChronoUnit.SECONDS`, `ChronoUnit.NANOS`).

Ejemplo:

```

1 LocalDateTime dateTime = LocalDateTime.of(2016, 7, 28, 14, 30);
2 LocalDateTime dateTime1 = dateTime.plus(1, ChronoUnit.DAYS).plus(1, ChronoUnit.HOURS); LocalDateTime dateTime2 = dateTime.minus(1,
3 ChronoUnit.DAYS).minus(1, ChronoUnit.HOURS);
4 System.out.println(dateTime); // 2016-07-28T14:30
5 System.out.println(dateTime1); // 2016-07-29T15:30
6 System.out.println(dateTime2); // 2016-07-27T13:30

```

En el siguiente ejemplo, se hace uso de `Period` y `Duration`:

```

1 LocalDateTime dateTime = LocalDateTime.of(2016, 7, 28, 14, 30);
2 LocalDateTime dateTime1 = dateTime.plus(Period.ofDays(1)).plus(Duration.ofHours(1));
3 LocalDateTime dateTime2 = dateTime.minus(Period.ofDays(1)).minus(Duration.ofHours(1));
4 System.out.println(dateTime); // 2016-07-28T14:30
5 System.out.println(dateTime1); // 2016-07-29T15:30
6 System.out.println(dateTime2); // 2016-07-27T13:30

```

Finalmente, haciendo uso de los métodos `plusX(long xToAdd)` o `minusX(long xToSubtract)`:

```

1 LocalDateTime dateTime = LocalDateTime.of(2016, 7, 28, 14, 30);
2 LocalDateTime dateTime1 = dateTime.plusDays(1).plusHours(1);
3 LocalDateTime dateTime2 = dateTime.minusDays(1).minusHours(1);
4 System.out.println(dateTime); // 2016-07-28T14:30
5 System.out.println(dateTime1); // 2016-07-29T15:30
6 System.out.println(dateTime2); // 2016-07-27T13:30

```

Además, métodos como `isBefore`, `isAfter`, `isEqual` están disponibles para comparar las siguientes clases `LocalDate`, `LocalTime` y `LocalDateTime`.

Ejemplo:

```

1 LocalDate date1 = LocalDate.of(2016, 7, 28);
2 LocalDate date2 = LocalDate.of(2016, 7, 29);
3 boolean isBefore = date1.isBefore(date2); //true
4 boolean isAfter = date2.isAfter(date1); //true
5 boolean isEqual = date1.isEqual(date2); //false

```

### 17.2.7. Formatos

Cuando se trabaja con fechas, en ocasiones se requiere de un formato personalizado. Podemos usar el método `ofPattern(String pattern)`, para definir un formato en particular.

Para utilizar `DateTimeFormatter.ofPattern` debemos importar la clase con `import java.time.format.DateTimeFormatter;`

Ejemplo:

```
1 LocalDate mifecha = LocalDate.of(2016, 7, 25);
2 String fechaTexto=mifecha.format(DateTimeFormatter.ofPattern("eeee', ' dd 'de' MMMM 'del' u"));
3 System.out.println("La fecha es: "+fechaTexto); // La fecha es: lunes, 25 de julio del 2016
```

El patrón del formato se realiza en función a la siguiente tabla de símbolos:

| Símbolo | Descripción              | Salida              |
|---------|--------------------------|---------------------|
| y       | Año                      | 2004; 04            |
| D       | Día del Año              | 189                 |
| M       | Mes del Año              | 7; 07; Jul; July; J |
| d       | Día del Mes              | 10                  |
| w       | Semana del Año           | 27                  |
| E       | Día de la Semana         | Tue; Tuesday; T     |
| F       | Semana del Mes           | 3                   |
| a       | AM/PM                    | PM                  |
| K       | Hora AM/PM (0-11)        | 0                   |
| H       | Hora del día (0-23)      | 0                   |
| m       | Minutos de la hora       | 30                  |
| s       | Segundos del minuto      | 55                  |
| n       | Nanosegundos del Segundo | 987654321           |
| "       | Texto                    | 'Día de la semana'  |

#### 17.2.7.1. DÍA DE LA SEMANA

La función `getDayOfWeek()` devuelve un elemento del tipo `DayOfWeek` que corresponde el día de la semana de una fecha. Debemos importar la clase `java.time.DayOfWeek`.

Por ejemplo, el lunes será `DayOfWeek.MONDAY`.

Ejemplo:

```
1 LocalDate lafecha = LocalDate.of(2016, 7, 25);
2 if (lafecha.getDayOfWeek().equals(DayOfWeek.SATURDAY)) {
3 System.out.println("La fecha es Sábado");
4 } else {
5 System.out.println("La fecha NO es Sábado");
6 }
7 //La fecha NO es Sábado
```

## 17.3. Ejemplo Anexo UD05

### 17.3.1. Anexo1Wrappers

```

1 package es.martinezpenya.ejemplos.UD05;
2
3 public class Anexo1Wrappers {
4
5 public static void main(String[] args) {
6
7 // WRAPPERS
8 //Integer i1 = new Integer(42); // Obsoleto (deprecated)
9 Integer i1 = Integer.valueOf(42);
10 //Integer i2 = new Integer("42");// Obsoleto (deprecated)
11 Integer i2 = Integer.valueOf("42");
12 //Float f1 = new Float(3.14f);// Obsoleto (deprecated)
13 Float f1 = Float.valueOf(3.14f);
14 //Float f2 = new Float("3.14f");// Obsoleto (deprecated)
15 Float f2 = Float.valueOf("3.14f");
16
17 Integer y = Integer.valueOf(567); //Crea el objeto
18 y++; //Lo desenvuelve, incrementa y lo vuelve a envolver
19 System.out.println("Valor: " + y); //Imprime el valor del Objeto y
20
21 // VALUEOF
22 // Convierte el 101011 (base 2) a 43 y le asigna el valor al objeto Integer i1
23 Integer i3 = Integer.valueOf("101011", 2);
24 System.out.println(i3);
25
26 // Asigna 3.14 al objeto Float f3
27 Float f3 = Float.valueOf("3.14f");
28 System.out.println(f3);
29
30 // XXXVALUE
31 Integer i4 = 120; // Crea un nuevo objeto wrapper
32 byte b = i4.byteValue(); // Convierte el valor de i2 a un primitivo byte
33 short s1 = i4.shortValue(); // Otro de los métodos de Integer
34 double d = i4.doubleValue(); // Otro de los métodos xxxValue de Integer
35 System.out.println(s1); // Muestra 120 como resultado
36
37 Float f4 = 3.14f; // Crea un nuevo objeto wrapper
38 short s2 = f4.shortValue(); // Convierte el valor de f2 en un primitivo short
39 System.out.println(s2); // El resultado es 3 (truncado, no redondeado)
40
41 // PARSEXXX
42 double d4 = Double.parseDouble("3.14"); // Convierte un String a primitivo
43 System.out.println("d4 = " + d4); // El resultado será d4 = 3.14
44 long l2 = Long.parseLong("101010", 2); // un String binario a primitivo
45 System.out.println("l2 = " + l2); // El resultado es L2 42
46
47 // TOSTRING
48 Double d1 = Double.valueOf("3.14");
49 System.out.println("d1 = " + d1.toString()); // El resultado es d = 3.14
50 String d2 = Double.toString(3.14); // d2 = "3.14"
51 System.out.println("d2 = " + d2); // El resultado es d = 3.14
52 String s3 = "hex = " + Long.toHexString(254, 16); // s = "hex = fe"
53 System.out.println("s3 = " + s3); // El resultado es d = 3.14
54
55 // TOXXXSTRING
56 String s4 = Integer.toHexString(254); // Convierte 254 a hex
57 System.out.println("254 es " + s4); // Resultado: "254 es fe"
58 String s5 = Long.toOctalString(254); // Convierte 254 a octal
59 System.out.println("254(oct) = " + s5); // Resultado: "254(oct) = 376"
60 }
61 }
```

### 17.3.2. Anexo2Date

```

1 package UD05;
2
3 import java.util.Calendar;
4 import java.util.Date;
5 import java.util.GregorianCalendar;
6 import java.time.*;
7 import java.time.format.DateTimeFormatter;
8 import java.time.temporal.ChronoUnit;
9
10 public class Anexo2Date {
11
12 public static void main(String[] args) {
13
14 //Clase Date (java.util.Date)
15 Date fecha = new Date(2021, 8, 19);
16 System.out.println(fecha); //Mon Sep 19 00:00:00 CEST 3921
17 System.out.println(fecha.getTime()); //61590146400000
18
19 //Clase GregorianCalendar (java.util.Calendar y java.util.GregorianCalendar)
20 Calendar calendar = new GregorianCalendar(2021, 8, 19);
21 System.out.println(calendar.getTime()); //Sun Sep 19 00:00:00 CEST 2021
22
23 Date d = new Date();
24 GregorianCalendar c = new GregorianCalendar();
25 System.out.println("Fecha: " + d); //Fecha: Thu Aug 19 20:06:14 CEST 2021
26 System.out.println("Info: " + c); //Info: java.util.GregorianCalendar[time=1629396374723,
27 //areFieldsSet=true,areAllFieldsSet=true,lenient=true,zone=sun.util.calendar.ZoneInfo
28 //[id=Europe/Madrid],offset=3600000,dstSavings=3600000,useDaylight=true,transitions=163,
29 //lastRule=java.util.SimpleTimeZone[id=Europe/Madrid,offset=3600000,dstSavings=3600000,
30 //useDaylight=true,startYear=0,startMode=2,startMonth=2,startDay=-1,startDayOfWeek=1,
31 //startTime=3600000,startTimeMode=2,endMode=2,endMonth=9,endDay=-1,endDayOfWeek=1,
32 //endTime=3600000,endTimeMode=2],firstDayOfWeek=2,minimalDaysInFirstWeek=4,ERA=1,
33 //YEAR=2021,MONTH=7,WEEK_OF_YEAR=33,WEEK_OF_MONTH=3,DAY_OF_MONTH=19,DAY_OF_YEAR=231,
34 //DAY_OF_WEEK=5,DAY_OF_WEEK_IN_MONTH=3,AM_PM=1,HOUR=8,HOUR_OF_DAY=20,MINUTE=6,SECOND=14,
35 //MILLISECOND=723,ZONE_OFFSET=3600000,DST_OFFSET=3600000]
36 c.setTime(d);
37 System.out.print(c.get(Calendar.DAY_OF_MONTH));
38 System.out.print("/");
39 System.out.print(c.get(Calendar.MONTH) + 1);
40 System.out.print("/");
41 System.out.println(c.get(Calendar.YEAR) + 1); //19/8/2022
42
43 //LocalDate, LocalTime, LocalDateTime, Duration y Period (java.time.*)
44 //Localdate d = new LocalDate(); //NO compila
45 LocalDate date = LocalDate.of(1989, 11, 11); //1989-11-11
46 System.out.println(date.getYear()); //1989
47 System.out.println(date.getMonth()); //NOVEMBER
48 System.out.println(date.getDayOfMonth()); //11
49 date = LocalDate.now();
50 System.out.println(date); //2021-08-19
51
52 LocalTime time = LocalTime.of(5, 30, 45, 35); //05:30:45:35
53 System.out.println(time.getHour()); //5
54 System.out.println(time.getMinute()); //30
55 System.out.println(time.getSecond()); //45
56 System.out.println(time.getNano()); //35
57 time = LocalTime.now();
58 System.out.println(time); //20:13:53.118044
59
60 LocalDateTime dateTime = LocalDateTime.of(1989, 11, 11, 5, 30, 45, 35);
61
62 LocalDate date2 = LocalDate.of(1989, 11, 11);
63 LocalTime time2 = LocalTime.of(5, 30, 45, 35);
64 LocalDateTime dateTime1 = LocalDateTime.of(date, time);
65 LocalDateTime dateTime2 = LocalDateTime.now();
66
67 LocalTime localTime1 = LocalTime.of(12, 25);
68 LocalTime localTime2 = LocalTime.of(17, 35);
69 Duration duration1 = Duration.between(localTime1, localTime2);
70 System.out.println(duration1); //PT5H10M
71 System.out.println(duration1.toDays()); //0
72
73 LocalDateTime localDateTime1 = LocalDateTime.of(2016, Month.JULY, 18, 14, 13);
74 LocalDateTime localDateTime2 = LocalDateTime.of(2016, Month.JULY, 20, 12, 25);
75 Duration duration2 = Duration.between(localDateTime1, localDateTime2);
76 System.out.println(duration2); //PT46H12M
77 System.out.println(duration2.toDays()); //1
78
79 Duration duracion3 = Duration.ofDays(1);
80 System.out.println(duracion3); //PT24H
81 System.out.println(duracion3.toDays()); //1
82
83 LocalDate localDate1 = LocalDate.of(2016, 7, 18);
84 LocalDate localDate2 = LocalDate.of(2016, 7, 20);
85 Period periodo1 = Period.between(localDate1, localDate2);
86 System.out.println(periodo1); //P2D

```

```

1 Period periodo2 = Period.of(1, 2, 3);
2 System.out.println(periodo2); //P1Y2M3D
3
4 Period periodo3 = Period.ofYears(1);
5 System.out.println(periodo3); //P1Y
6
7 //CHRONOUNIT (java.time.temporal.ChronoUnit)
8 LocalDate fechaInicio = LocalDate.of(2016, 7, 18);
9 LocalDate fechaFin = LocalDate.of(2016, 7, 20);
10 // Calculamos el tiempo transcurrido entre las dos fechas
11 // con la clase ChronoUnit y la unidad temporal en la que
12 // queremos que nos lo devuelva, en este caso DAYS.
13 long tiempo = ChronoUnit.DAYS.between(fechaInicio, fechaFin);
14 System.out.println(tiempo); //2
15
16 //Introducir fecha por teclado (java.time.format.DateTimeFormatter)
17 DateTimeFormatter formato = DateTimeFormatter.ofPattern("d/MM/yyyy");
18 String fechaCadena = "16/08/2016";
19 LocalDate mifecha = LocalDate.parse(fechaCadena, formato);
20 System.out.println(formato.format(mifecha)); //16/08/2016
21
22 //Manipulación
23 LocalDate fec = LocalDate.of(2016, 7, 25);
24 LocalDate fec1 = fec.withYear(2017);
25 LocalDate fec2 = fec.withMonth(8);
26 LocalDate fec3 = fec.withDayOfMonth(27);
27 System.out.println(date); //2016-07-25
28 System.out.println(fec1); //2017-07-25
29 System.out.println(fec2); //2016-08-25
30 System.out.println(fec3); //2016-07-27
31
32 LocalTime tim = LocalTime.of(14, 30, 35);
33 LocalTime tim1 = tim.withHour(20);
34 LocalTime tim2 = tim.withMinute(25);
35 LocalTime tim3 = tim.withSecond(23);
36 LocalTime tim4 = tim.withNano(24);
37 System.out.println(tim); //14:30:35
38 System.out.println(tim1); //20:30:35
39 System.out.println(tim2); //14:25:35
40 System.out.println(tim3); //14:30:23
41 System.out.println(tim4); //14:30:35.00000024
42
43 LocalDateTime dateTim = LocalDateTime.of(2016, 7, 25, 22, 11, 30);
44 LocalDateTime dateTim1 = dateTim.withYear(2017);
45 LocalDateTime dateTim2 = dateTim.withMonth(8);
46 LocalDateTime dateTim3 = dateTim.withDayOfMonth(27);
47 LocalDateTime dateTim4 = dateTim.withHour(20);
48 LocalDateTime dateTim5 = dateTim.withMinute(25);
49 LocalDateTime dateTim6 = dateTim.withSecond(23);
50 LocalDateTime dateTim7 = dateTim.withNano(24);
51 System.out.println(dateTim); //2016-07-25T22:11:30
52 System.out.println(dateTim1); //2017-07-25T22:11:30
53 System.out.println(dateTim2); //2016-08-25T22:11:30
54 System.out.println(dateTim3); //2016-07-27T22:11:30
55 System.out.println(dateTim4); //2016-07-25T20:11:30
56 System.out.println(dateTim5); //2016-07-25T22:25:30
57 System.out.println(dateTim6); //2016-07-25T22:11:23
58 System.out.println(dateTim7); //2016-07-25T22:11:30.00000024
59
60 //OPERACIONES
61 LocalDate date3 = LocalDate.of(2016, 7, 18);
62 LocalDate date3PlusOneDay = date3.plus(1, ChronoUnit.DAYS);
63 LocalDate date3MinusOneDay = date3.minus(1, ChronoUnit.DAYS);
64 System.out.println(date3); // 2016-07-18
65 System.out.println(date3PlusOneDay); // 2016-07-19
66 System.out.println(date3MinusOneDay); // 2016-07-17
67
68 LocalDate date4 = LocalDate.of(2016, 7, 18);
69 LocalDate date4PlusOneDay = date4.plus(Period.ofDays(1));
70 LocalDate date4MinusOneDay = date4.minus(Period.ofDays(1));
71 System.out.println(date4); // 2016-07-18
72 System.out.println(date4PlusOneDay); // 2016-07-19
73 System.out.println(date4MinusOneDay); // 2016-07-17
74
75 LocalDate date5 = LocalDate.of(2016, 7, 18);
76 LocalDate date5PlusOneDay = date5.plusDays(1);
77 LocalDate date5MinusOneDay = date5.minusDays(1);
78 System.out.println(date5); // 2016-07-18
79 System.out.println(date5PlusOneDay); // 2016-07-19
80 System.out.println(date5MinusOneDay); // 2016-07-17
81
82 LocalTime time3 = LocalTime.of(15, 30);
83 LocalTime time3PlusOneHour = time3.plus(1, ChronoUnit.HOURS);
84 LocalTime time3MinusOneHour = time3.minus(1, ChronoUnit.HOURS);
85 System.out.println(time3); // 15:30
86 System.out.println(time3PlusOneHour); // 16:30
87 System.out.println(time3MinusOneHour); // 14:30

```

```

1 LocalTime time4 = LocalTime.of(15, 30);
2 LocalTime time4PlusOneHour = time4.plus(Duration.ofHours(1));
3 LocalTime time4MinusOneHour = time4.minus(Duration.ofHours(1));
4 System.out.println(time4); // 15:30
5 System.out.println(time4PlusOneHour); // 16:30
6 System.out.println(time4MinusOneHour); // 14:30
7
8 LocalTime time5 = LocalTime.of(15, 30);
9 LocalTime time5PlusOneHour = time5.plusHours(1);
10 LocalTime time5MinusOneHour = time5.minusHours(1);
11 System.out.println(time5); // 15:30
12 System.out.println(time5PlusOneHour); // 16:30
13 System.out.println(time5MinusOneHour); // 14:30
14
15 LocalDateTime dateTime3 = LocalDateTime.of(2016, 7, 28, 14, 30);
16 LocalDateTime dateTime4 = dateTime3.plus(1, ChronoUnit.DAYS).plus(1, ChronoUnit.HOURS);
17 LocalDateTime dateTime5 = dateTime3.minus(1, ChronoUnit.DAYS).minus(1, ChronoUnit.HOURS);
18 System.out.println(dateTime3); // 2016-07-28T14:30
19 System.out.println(dateTime4); // 2016-07-29T15:30
20 System.out.println(dateTime5); // 2016-07-27T13:30
21
22 LocalDateTime dateTime6 = LocalDateTime.of(2016, 7, 28, 14, 30);
23 LocalDateTime dateTime7 = dateTime6.plus(Period.ofDays(1)).plus(Duration.ofHours(1));
24 LocalDateTime dateTime8 = dateTime6.minus(Period.ofDays(1)).minus(Duration.ofHours(1));
25 System.out.println(dateTime6); // 2016-07-28T14:30
26 System.out.println(dateTime7); // 2016-07-29T15:30
27 System.out.println(dateTime8); // 2016-07-27T13:30
28
29 LocalDateTime dateTime9 = LocalDateTime.of(2016, 7, 28, 14, 30);
30 LocalDateTime dateTime10 = dateTime9.plusDays(1).plusHours(1);
31 LocalDateTime dateTime11 = dateTime9.minusDays(1).minusHours(1);
32 System.out.println(dateTime9); // 2016-07-28T14:30
33 System.out.println(dateTime10); // 2016-07-29T15:30
34 System.out.println(dateTime11); // 2016-07-27T13:30
35
36 LocalDate dat1 = LocalDate.of(2016, 7, 28);
37 LocalDate dat2 = LocalDate.of(2016, 7, 29);
38 boolean isBefore = dat1.isBefore(dat2); //true
39 boolean isAfter = date2.isAfter(dat1); //true
40 boolean isEqual = dat1.isEqual(dat2); //false
41
42 //Formatos (java.time.format.DateTimeFormatter)
43 LocalDate mifecha2 = LocalDate.of(2016, 7, 25);
44 String fechaTexto = mifecha2.format(DateTimeFormatter.
45 ofPattern("eeee', ' dd 'de' MMMM 'del' yyyy"));
46 System.out.println("La fecha es: " +
47 fechaTexto); // La fecha es: lunes, 25 de julio del 2016
48
49 //DAYOFWEEK
50 LocalDate lafecha = LocalDate.of(2016, 7, 25);
51 if (lafecha.getDayOfWeek().equals(DayOfWeek.SATURDAY)) {
52 System.out.println("La fecha es Sábado");
53 } else {
54 System.out.println("La fecha NO es Sábado");
55 }
56 //La fecha NO es Sábado
57 }
58 }
```

⌚12 de enero de 2026

## 18. 6.3 Ejercicios de la UD05

### 18.1. Ejercicios

#### 18.1.1. Paquete: UD05.\_1.gestionEmpleados

- Una empresa quiere hacer una gestión informatizada básica de sus empleados. Para ello, de cada empleado le interesa:
  - Nombre (String)
  - DNI (String)
  - Año de ingreso (número entero)
  - Sueldo bruto anual (número real)
- Diseñar una clase Java `Empleado`, que contenga los atributos (privados) que caracterizan a un empleado e implemente los métodos adecuados para:
  - Crear objetos de la clase: **Constructor** que reciba todos los datos del empleado a crear.
  - Consultar el valor de cada uno de sus atributos. (**Consultores o getters**)
  - `public int antiguedad()`. Devuelve el número de años transcurridos desde el ingreso del empleado en la empresa. Si el año de ingreso fuera posterior al de la fecha actual, devolverá 0. Para obtener el año actual puedes usar:
  - `java int anyoActual = Calendar.getInstance().get(Calendar.YEAR);`
  - `public void incrementarSueldo(double porcentaje)`. Incrementa el sueldo del empleado en un porcentaje dado (expresado como una cantidad real entre 0 y 100).
  - `public String toString()`. Devuelve un `String` con los datos del empleado, de la siguiente forma:

```

1 Nombre: Juan González
2 Dni: 545646556K
3 Año de ingreso: 1998
4 Sueldo bruto anual: 20000 €

```

- `public boolean equals(Object o)`. Método para comprobar si dos empleados son iguales. Dos empleados se consideran iguales si tienen el mismo DNI.
- `public int compareTo(Empleado o)`. Se considera menor o mayor el empleado que tiene menor o mayor DNI (el mismo criterio que al comparar dos strings).
- Método estático `public static double calcularIRPF(double salario)`. Determina el % de IRPF que corresponde a un salario (mensual) determinado, según la siguiente tabla:

| Desde salario (incluido) | Hasta salario (no incluido) | % IRPF |
|--------------------------|-----------------------------|--------|
| 0                        | 800                         | 3      |
| 800                      | 1000                        | 10     |
| 1000                     | 1500                        | 15     |
| 1500                     | 2100                        | 20     |
| 2100                     | infinito                    | 30     |

- Diseñar una clase Java `TestEmpleado` que permita probar la clase `Empleado` y sus métodos. Para ello se desarrollará el método `main` en el que:
  - Se crearán dos empleados utilizando los datos que introduzca el usuario.
  - Se incrementará el sueldo un 20 % al empleado que menos cobre.
  - Se incrementará el sueldo un 10% al empleado más antiguo.
  - Muestra el IRPF que correspondería a cada empleado.
  - Para comprobar que las operaciones se realizan correctamente, muestra los datos de los empleados tras cada operación.
- Diseñar una clase `Empresa`, que permita almacenar el nombre de la empresa y la información de los empleados de la misma (máximo 10 empleados) en un array. Para ello, se utilizarán tres atributos: `nombre`, `plantilla` (array de empleados) y `numEmpleados` (número de empleados que tiene la empresa). En esta clase, se deben implementar los métodos:
  - `public Empresa (String nombre)`. Constructor de la clase. Crea la empresa con el nombre indicado y sin empleados.

- `public void contratar(Empleado e) throws PlantillaCompletaException`. Añade el empleado indicado a la plantilla de la empresa, siempre que quepa en el array. Si no cabe, se lanzará la excepción `PlantillaCompletaException`.
  - `public void despedir(Empleado e) throws ElementoNoEncontradoException`. Elimina el empleado indicado de la plantilla. Si no existe en la empresa, se lanza `ElementoNoEncontradoException`.
  - `public void subirTrienio (double porcentaje)` Subir el sueldo, en el porcentaje indicado, a todos los empleados cuya antigüedad sea exactamente tres años.
  - `public String toString()`. Devuelve un `String` con el nombre de la empresa y la información de todos los empleados. La información de los distintos empleados debe estar separada por saltos de línea.
5. Diseñar una clase Java `TestEmpresa` que permita probar la clase `Empresa` y sus métodos. Para ello, desarrolla el método `main` y en él ....:
- Crea una empresa, de nombre "DAMCarlet".
  - Contrata a varios empleados (con el nombre, DNI, etc. que quieras).
  - Usa el método `subirTrienio` para subir un 10% el salario de los empleados que cumplen un trienio en el año actual.
  - Despide a alguno de los empleados.
  - Trata de despedir a algún empleado que no exista en la empresa.
  - Muestra los datos de la empresa siempre que sea necesario para comprobar que las operaciones se realizan de forma correcta.

### 18.1.2. Paquete: UD05.\_2.gestionHospital

1. Se desea realizar una aplicación para gestionar el ingreso y el alta de pacientes de un hospital. Una de las clases que participará en la aplicación será la clase `Paciente`, que se detalla a continuación :
2. La clase `Paciente` permite representar un paciente mediante los atributos: `nombre` (cadena), `edad` (entero), `estado` (entero entre 1 -más grave- y 5 -menos grave-, 6 si está curado), y con las siguientes operaciones:
  - `public Paciente (String n, int e)`. Constructor de un objeto `Paciente` de nombre `n`, de `e` años y cuyo estado es un valor aleatorio entre 1 y 5.
  - `public int getEdad()`. Consultor que devuelve edad.
  - `public int getEstado()`. Consultor que devuelve estado.
  - `public void mejorar()`. Modificador que incrementa en uno el estado del paciente (mejora al paciente)
  - `public void empeorar()`. Modificador que decrementa en uno el estado del paciente (empeora al paciente)
  - `public String toString()`. Transforma el paciente en un `String`. Por ejemplo,

```
1 Pepe Pérez 46 5
```

- `public int compareTo(Paciente o)`. Permite comparar dos pacientes. Se considera menor el paciente más leve. A igual gravedad, se considera menor el paciente más joven. Ejemplo:
- Teniendo a David 40 3, Pepe 25 3 y Juan 35 5 :

```
1 David.compareTo(Juan) = 2
2 Juan.compareTo(Pepe) = -2
3 David.compareTo(Pepe) = 15
```

3. Diseñar una clase Java `TestPaciente` que permita probar la clase `Paciente` y sus métodos. Para ello se desarrollará el método `main` en el que:
  - Se crearán dos pacientes: "Antonio" de 20 años y "Miguel" de 30 años.
  - Imprimir el estado inicial de los dos pacientes.
  - Mostrar los datos del que se considere menor (según el criterio de `compareTo` de la clase `Paciente`).
  - Aplicar "mejoras" al paciente más grave hasta que los dos pacientes tengan el mismo estado.
  - Imprimir el estado final de los dos pacientes.
4. La clase **Hospital** contiene la información de las camas de un hospital, así como de los pacientes que las ocupan. Un Hospital tiene un número máximo de camas `MAXC = 200` y para representarlas se utilizará un array (llamado `listacamas`) de objetos de tipo `Paciente` junto con un atributo (`numLibres`) que indique el número de camas libres del hospital en un momento dado. El número de cada cama coincide con su posición en el array de pacientes (la posición 0 no se utiliza), de manera que `listacamas[i]` es el Paciente que ocupa la cama `i` o es `null` si la cama está libre. Las operaciones de esta clase son:
  - `public Hospital()`. Constructor de un hospital. Cuando se crea un hospital, todas las camas están libres.
  - `public int getNumLibres()`. Consultor del número de camas libres.

- `public boolean hayLibres()`. Devuelve true si en el hospital hay camas libres y devuelve false en caso contrario.
- `public int primeraLibre()`. Devuelve el número de la primera cama libre del array `listaCamas` si hay camas libres o devuelve un 0 si no las hay.
- `public void ingresarPaciente(String n, int e)` throws `HospitalLlenoException` Si hay camas libres, la primera de ellas (la de número menor) pasa a estar ocupada por el paciente de nombre `n` y edad `e`. Si no hay camas libres, lanza una excepción.
- `private void darAltaPaciente(int i)`. La cama `i` del hospital pasa a estar libre. (Afectará al número de camas libres)
- `public void darAltas()`. Se mejora el estado (método `mejorar()` de `Paciente`) de cada uno de los pacientes del hospital y a aquellos pacientes sanos (cuyo estado es 6) se les da el alta médica (invocando al método `darAltaPaciente`).
- `public String toString()`. Devuelve un `String` con la información de las camas del hospital. Por ejemplo,

```

1 1 María Medina 30 4
2 2 Pepe Pérez 46 5
3 3 libre
4 4 Juan López 50 1
5 5 libre
6 ...
7 199 Andrés Sánchez 29 3

```

5. En la clase `GestorHospital` se probará el comportamiento de las clases anteriores. El programa deberá:

- Crear un hospital.
- Ingresar a cinco pacientes con los datos simulados introducidos directamente en el programa.
- Realizar el proceso de `darAltas` mientras que el número de habitaciones libres del hospital no llegue a una cantidad (por ejemplo 198).
- Mostrar los datos del hospital cuando se considere oportuno para comprobar la corrección de las operaciones que se hacen.

#### 18.1.3. Paquete: UD05.\_3.contrarreloj

1. Se quiere realizar una aplicación para registrar las posiciones y tiempos de llegada en una carrera ciclista contrarreloj.
2. La clase `Corredor` representa a un participante en la carrera. Sus atributos son el dorsal (entero), el nombre (`String`) y el tiempo en segundos (`double`) que le ha costado completar el recorrido. Los métodos con los que cuenta son:
  - `public Corredor(int d, String n)`. Constructor a partir del dorsal y el nombre. Por defecto el tiempo tardado es 0
  - `public double getTiempo()`. Devuelve el tiempo tardado por el corredor
  - `public int getDorsal()`. Devuelve el dorsal del corredor
  - `public String getNombre()`. Devuelve el nombre del corredor
  - `public void setTiempo(double t)` throws `IllegalArgumentException`. Establece el tiempo tardado por el corredor. Lanzará la excepción si el tiempo indicado es negativo.
  - `public void setTiempo(double t1, double t2)` throws `IllegalArgumentException`. Establece el tiempo tardado por el corredor. `t1` indica la hora de comienzo y `t2` la hora de finalización (expresadas en segundos). La diferencia en segundos entre los dos datos servirá para establecer el tiempo tardado por el `corredor`. Lanzará la excepción si el tiempo resultante es negativo
  - `public String toString()`. Devuelve un `String` con los datos del corredor, de la forma:

```

1 (234) - Juan Ramirez - 2597 segundos

```

- `public boolean equals(Object o)`. Devuelve true si los corredores tienen el mismo dorsal y false en caso contrario
- `public int compareTo (Corredor o)`. Un corredor es menor que otro si tiene menor dorsal.
- `public static int generarDorsal()`. Devuelve un número de dorsal generado secuencialmente. Para ello la clase hará uso de un atributo `static int siguienteDorsal` que incrementará cada vez que se genere un nuevo dorsal.
3. Diseñar una clase Java `TestCorredor` que permita probar la clase `Corredor` y sus métodos. Para ello se desarrollará el método `main` en el que:
  - Se crearán dos corredores: El nombre lo indicará el usuario mientras que el dorsal se generará utilizando el método `generarDorsal()` de la clase.
  - Se establecerá el tiempo de llegada del primer corredor a 300 segundos y el del segundo a 400.
  - Se mostrarán los datos de ambos corredores (`toString`)
4. La clase `ListaCorredores` permite representar a un conjunto de corredores. En la lista, como máximo habrá 200 corredores, aunque puede haber menos de ese número. Se utilizará un array, llamado `lista`, de 200 elementos junto con una propiedad `numCorredores` que permita saber cuantos corredores hay realmente. Métodos:
  - `public ListaCorredores()`. Constructor. Crea la lista de corredores, inicialmente vacía.

- `public void anyadir(Corredor c) throws ElementoDuplicadoException`. Añade un corredor al final de la lista de corredores (pero lo más al principio posible del array), siempre y cuando el corredor no esté ya en la lista, en cuyo caso se lanzará `ElementoDuplicadoException`
- `public void insertarOrdenado(Corredor c)`. Inserta un corredor en la posición adecuada de la lista de manera que esta se mantenga ordenada crecientemente por el tiempo de llegada. Para poder realizar la inserción debe averiguar la posición que debe ocupar el nuevo elemento y, antes de añadirlo al array, desplazar el elemento que ocupa esa posición y todos los posteriores, una posición a la derecha.
- `public Corredor quitar(int dorsal) throws ElementoNoEncontradoException`. Quita de la lista al corredor cuyo dorsal se indica. El array debe mantenerse compacto, es decir, todos los elementos posteriores al eliminado deben desplazarse una posición a la izquierda. El método devuelve el Corredor quitado de la lista. Si no se encuentra se lanza `ElementoNoEncontradoException`.
- `public String toString()` Devuelve un `String` con la información de la lista de corredores. Los minutos aparecerán formateados con 2 decimales. Por ejemplo:

```

1 Posición: 0
2 Dorsal: 234
3 Nombre: Juan Ramirez
4 Tiempo: 25.97 minutos
5
6 Posición: 1
7 Dorsal: 26
8 Nombre: José González
9 Tiempo: 29.70 minutos

```

5. Clase `contrarreloj`) Realizar un programa que simule una contrarreloj. Para llevar el control de una carrera contrarreloj se mantienen dos listas de corredores (dos objetos de tipo `ListaCorredores`):
- (`hanSalido`) Una con los que han salido, que tiene a los corredores por orden de salida. El atributo `tiempo` de estos corredores será 0. Para que los corredores se mantengan por orden de salida, se añadirán a la lista utilizando el método `añadir`.
  - (`hanLlegado`) Otra con los corredores que han llegado a la meta. A medida que los corredores llegan a la meta se les extrae de la primera lista, se les asigna un tiempo y se les inserta ordenadamente en esta segunda lista.
  - En el método `main` realizar un programa que muestre un menú con las siguientes opciones:
    - `Salida`: Para registrar que una corredor ha comenzado la contrarreloj y sale de la línea de salida. Solicitud al usuario el nombre de un corredor y su dorsal, y lo añade a la lista de corredores que han salido.
    - `Llegada`: Para registrar que un corredor ha llegado a la meta. Solicitud al usuario el dorsal de un corredor y el tiempo de llegada (en segundos). Quita al corredor de la lista de corredores que `hanSalido`, le asigna el tiempo que ha tardado y lo inserta (ordenadamente) en la lista de corredores que `hanLlegado`
    - `Clasificación`: Muestra la lista de corredores que `hanLlegado`. Dado que esta lista está ordenada por tiempo, mostrarla por pantalla nos da la clasificación.
    - `Salir`: Sale del programa

#### 18.1.4. Paquete: UD05.\_4.reservasLibreria

1. Una librería quiere proporcionar a sus clientes el siguiente servicio:

Cuando un cliente pide un libro y la librería no lo tiene, el cliente puede hacer una reserva de manera que cuando lo reciban en la librería le avisen por teléfono.

De cada reserva se almacena:

- `Nif del cliente (String)`
- `Nombre del cliente (String)`
- `Teléfono del cliente (String)`
- `Código del libro reservado. (entero)`
- `Número de ejemplares (entero)`

2. Diseñar la clase `Reserva`, de manera que contemple la información descrita e implementar:

- `public Reserva(String nif, String nombre, String tel, int codigo, int ejemplares)`. Constructor que recibe todos los datos de la reserva.
- `public Reserva(String nif, String nombre, String tel, int codigo)`. Constructor que recibe los datos del cliente y el código del libro. Establece el número de ejemplares a uno.
- Consultores de todos los atributos.
- `public void setEjemplares(int ejemplares)`. Modificador del número de ejemplares. Establece el número de ejemplares al valor indicado como parámetro.

- `public String toString()` que devuelva un `String` con los datos de la reserva
  - `public boolean equals(Object o)`. Dos reservas son iguales si son del mismo cliente y reservan el mismo libro.
  - `public int compareTo(Object o)`. Es menor la reserva cuyo código de libro es menor. El parámetro es de tipo `Object` así que revisa si debes hacer alguna "adaptación".
3. Diseñar una clase Java `TestReservas` que permita probar la clase `Reserva` y sus métodos. Para ello se desarrollará el método `main` en el que:
- Se creen dos reservas con los datos que introduce el usuario. Las reservas no pueden ser iguales (`equals`). Si la segunda reserva es igual a la primera se pedirá de nuevo los datos de la segunda al usuario.
  - Se incremente en uno el número de ejemplares de ambas reservas.
  - Se muestre la menor y a continuación la mayor.
  - Mostrar el listado de reservas cada vez que consideres oportuno.
4. Diseñar una clase `ListaReservas` que implemente una lista de reservas. Como máximo puede haber 100 reservas en la lista. Se utilizará un array de `Reservas` que ocuparemos a partir de la posición 0 y un atributo que indique el número de reservas. Las reservas existentes ocuparán las primeras posiciones del array (sin espacios en blanco). Implementar los siguientes métodos:
- `public void reservar (String nif, String nombre, String telefono, int libro, int ejemplares) throws ListaLlenaException, ElementoDuplicadoException`: Crea una reserva y la añade a la lista. Lanza `ElementoDuplicadoException` si la reserva ya estaba en la lista. Lanza `ListaLlenaException` si la lista de reservas está llena.
  - `public void cancelar (String nif, int libro) throws ElementoNoEncontradoException`: Dado un nif de cliente y un código de libro, anular la reserva correspondiente. Lanzar `ElementoNoEncontradoException` si la reserva no existe.
  - `public String toString()`: Devuelve un `String` con los datos de todas las reservas de la lista.
  - `public int numEjemplaresReservadosLibro (int codigo)`: Devuelve el número de ejemplares que hay reservados en total de un libro determinado.
  - `public void reservasLibro (int codigo)`: Dado un código de libro, muestra el nombre y el teléfono de todos los clientes que han reservado el libro.
5. Realizar un programa `GestionReservas` que, utilizando un menú, permita:
- Realizar reserva. Permite al usuario realizar una reserva.
  - Anular reserva: Se anula la reserva que indique el usuario (Nif de cliente y código de libro).
  - Pedido: El usuario introduce un código de libro y el programa muestra el nº de reservas que se han hecho del libro. Esta opción de menú le resultará útil al usuario para poder hacer el pedido de un libro determinado.
  - Recepción: Cuando el usuario recibe un libro quiere llamar por teléfono a los clientes que lo reservaron. Solicitar al usuario un código de libro y mostrar los datos (nombre y teléfono) de los clientes que lo tienen reservado.

#### **18.1.5. Paquete: UD05.\_5.gestorCorreoElectronico**

1. Queremos realizar la parte de un programa de correo electrónico que gestiona la organización de los mensajes en distintas carpetas. Para ello desarrollaremos:
2. La clase `Mensaje`. De un mensaje conocemos:
- `Codigo (int)` Número que permite identificar a los mensajes.
  - `Emisor (String)`: email del emisor.
  - `Destinatario (String)`: email del destinatario.
  - `Asunto (String)`
  - `Texto (String)`
- Desarrollar los siguientes métodos:
- Constructor que reciba todos los datos, excepto el código, que se generará automáticamente (nº consecutivo. Ayuda: utiliza una variable de clase (`static`))
  - Consultores de todos los atributos.
  - `public boolean equals(Object o)`. Dos mensajes son iguales si tienen el mismo código.
  - `public static boolean validarEMail(String email)`: Método estático que devuelve true o false indicando si la dirección de correo indicada es válida o no. Una dirección es válida si tiene la forma `direccion@subdominio.dominio`
  - `public String toString()`
3. Con la clase `TestCorreo` probaremos las clases y métodos desarrollados.
- Crea varios mensajes con los datos que introduzca el usuario y muéstralos por pantalla.
  - Prueba el método `validarEMail` de la clase `Mensaje` con las direcciones siguientes (solo la primera es correcta):

- tuCorreo@gmail.com
- tuCorreogmail.com
- tuCorreo@gmail
- tuCorreo.com@gmail

4. La clase `Carpeta`, cada carpeta tiene un nombre y una lista de Mensajes. Para ello usaremos un array con capacidad para 100 mensajes y un atributo que indique el número de mensajes que contiene la carpeta. Además se implementarán los siguientes métodos:

- `public Carpeta(String nombre)`: Constructor. Dado un nombre, crea la carpeta sin mensajes.
- `public void anyadir(Mensaje m)`: Añade a la carpeta el mensaje indicado.
- `public void borrar(Mensaje m) throws ElementoNoEncontradoException`: Borra de la carpeta el mensaje indicado. Lanza la excepción si el mensaje no existe.
- `public Mensaje buscar(int codigo) throws ElementoNoEncontradoException`: Busca el mensaje cuyo código se indica. Si lo encuentra devuelve el mensaje, en caso contrario lanza la excepción.
- `public String toString()` que devuelva un `String` con el nombre de la carpeta y sus mensajes
- `public static void moverMensaje(Carpeta origen, Carpeta destino, int codigo) throws ElementoNoEncontradoException`: Método estático. Recibe dos Carpetas de correo y un código de mensaje y mueve el mensaje indicado de una carpeta a otra. Para ello buscará el mensaje en la carpeta origen. Si existe lo eliminará y lo añadirá a la carpeta de destino. Si el mensaje indicado no está en la carpeta de origen lanza `ElementoNoEncontradoException`.

5. Con la clase `TestCarpetas` probaremos las clases y métodos desarrollados:

- Crea dos carpetas de correo de nombre `Mensajes recibidos` y `Mensajes eliminados` respectivamente.
- Crea varios mensajes y añádelos a `Mensajes recibidos`.
- Mueve el mensaje de código 1 desde la `Mensajes recibidos` a `Mensajes eliminados`.
- Muestra el contenido de las carpetas antes y después de cada operación (añadir, mover,...)

#### **18.1.6. Paquete: UD05.\_6.juegoDeCartas**

1. Se está desarrollando una aplicación que usa una baraja de cartas. Para ello, se implementarán en Java las clases necesarias. 2. Una de ellas es la clase `Carta` que permite representar una carta de la baraja española. La información requerida para identificar una `Carta` es:

- su `palo` (oros, copas, espadas o bastos)
- su `valor` (un entero entre 1 y 12).

Para dicha clase, se pide:

- Definir 4 constantes, atributos de clase (estáticos) públicos enteros, para representar cada uno de los palos de la baraja (oros será el valor 0, COPAS el 1, ESPADAS el 2 y BASTOS el 3).
- Definir los atributos (privados): `palo` y `valor`.
- Escribir dos constructores: uno para construir una carta de forma aleatoria (sin parámetros) y otro para construir una carta de acuerdo a dos datos: su `palo` y su `valor` (si los datos son incorrectos se lanzará `IllegalArgumentException`)
- Escribir los métodos `consultores` y `modificadores` de los valores de los atributos.
- Escribir un método `compareTo` para comprobar si la carta actual es menor que otra carta dada. El criterio de ordenación es por palos (el menor es oros, después copas, a continuación espadas y, finalmente, bastos) y dentro de cada palo por valor (1, 2, ..., 12).
- Escribir un método `equals` para comprobar la igualdad de dos cartas. Dos cartas son iguales si tienen el mismo palo y valor.
- Escribir un método `sigPalo` para devolver una nueva carta con el mismo valor que el de la carta actual pero del palo siguiente, según la ordenación anterior y sabiendo que el siguiente al palo bastos es oros.
- Escribir un método `toString` para transformar en `String` la carta actual, con el siguiente formato: "valor de palo"; por ejemplo, "4 de oros" o "1 de bastos" (sobrescritura del método `toString` de `Object`).

3. Implementar una clase `JuegoCartas` con los métodos siguientes:

- Un método de clase (estático) `public static int ganadora( Carta c1, Carta c2, int triunfo)` que dados dos objetos `Carta` y un número entero representando el palo de triunfo (o palo ganador), determine cuál es la carta ganadora. El método debe devolver 0 si las dos cartas son iguales. En caso contrario, devolverá -1 cuando la primera carta es la ganadora y 1 si la segunda carta es la ganadora.
- Para determinar la carta ganadora se aplicarán las siguientes reglas:

- Si las dos cartas son del mismo palo, la carta ganadora es el as (valor 1) y, en el resto de casos, la carta ganadora es la de valor más alto (por ejemplo, "1 de oros" gana a "7 de oros", "5 de copas" gana a "2 de copas", "11 de bastos" gana a "7 de bastos").
- Si las dos cartas son de palos diferentes:
- Si el palo de alguna carta es el palo de triunfo, dicha carta es la ganadora.
- En otro caso, la primera carta siempre gana a la segunda.
- Un método `main` en el que se debe:
- Crear una `Carta` aleatoriamente y mostrar sus datos por pantalla.
- Generar aleatoriamente un entero en el rango [0..3] representando el palo de triunfo, y mostrar por pantalla a qué palo corresponde.
- Crear una `Carta` a partir de un palo y un valor dados (solicitados al usuario desde teclado), y mostrar sus datos por pantalla.
- Mostrar por pantalla la carta ganadora (invocando al método del apartado anterior con el objeto `Carta` del usuario).

## 18.2. Actividades

1. (WrapperDouble) Introducir por teclado un valor de tipo `double`, convertirlo en Wrapper e imprimirlo.
2. (StringAEntero) Introducir por teclado un valor numérico en un `String` y convertirlo en entero e imprimirlo.
3. (StringAWrapper) Introducir por teclado un valor numérico entero en un `String` y convertirlo en un `Wrapper` e imprimirlo.
4. (OperacionesBinarias) Introducir por teclado dos valores numéricos enteros y la operación que queremos realizar (`suma`, `resta` o `multiplicación`). Realizar la operación y mostrar el resultado en `Binario`, `Hexadecimal` y `Octal`.

Ejemplo de ejecución:

```

1 Introduce el primer valor numérico: 14
2 Introduce el segundo valor numérico: 4
3 Introduce la operación (suma, resta, multiplicacion): resta
4 EL RESULTADO:
5 en binario: 1010
6 en octal: 12
7 en hexadecimal: a

```

5. (SegundosDesde1970) Mostrar los segundos transcurridos desde el 1 de Enero de 1970 a las 0:00:00 hasta hoy.
6. (FormatosFechaHora) Mostrar la fecha y hora de hoy con los siguientes formatos (para todos los ejemplos se supone que hoy es 26 de agosto de 2021 a las 17 horas 16 minutos y 8 segundos, tu deberás mostrar la fecha y hora de tu sistema en el momento de ejecución):
  - a) August 26, 2021, 5:16 pm
  - b) 08.26.21
  - c) 26, 8, 2021
  - d) 20210826
  - e) 05-16-08, 26-08-21
  - f) Thu Aug 26 17:16:08
  - g) 17:16:08
7. (ValidarFecha) Introducir un día, un mes y un año y verificar si es una fecha correcta.

```

1 Introduce un dia para la fecha: 29
2 Introduce un mes para la fecha: 2
3 Introduce un año para la fecha: 2022
4 LA FECHA ES INCORRECTA
5
6 Introduce un dia para la fecha: 29
7 Introduce un mes para la fecha: 2
8 Introduce un año para la fecha: 2020
9 LA FECHA ES CORRECTA

```

8. (DiasEntreFechas) Introducir dos fechas e indicar los días transcurridos entre las dos fechas.

```

1 Introduce la fecha inicial con formato dd/mm/yyyy: 01/02/2021
2 Introduce la fecha final con formato dd/mm/yyyy: 15/03/2022
3 La fecha inicial es: 1/2/2021
4 La fecha final es: 15/3/2022
5 Entre la fecha inicial y la final hay un periodo de: P1Y1M14D
6 dias: 14
7 meses: 1
8 años: 1

```

9. (PagosPlazos) Introducir una fecha y devolver las fecha de los pagos a 30, 60 y 90 días.
10. (CompararFechas) Introducir tres fechas e indicar la mayor y a menor.
11. (FechaActualComparacion) Introducir el día, mes, año. Crear una fecha a partir de los datos introducidos y comprobar e indicar si se trata de la fecha actual, si es una fecha pasada o una fecha futura.
12. (EdadEmpleado) Introducir una fecha de nacimiento de un empleado e indicar cuántos años tiene el empleado.
13. (ProductoCaducado) Introducir la fecha de caducidad de un producto e indicar si el producto está o no caducado. El valor por defecto será la fecha actual y solo se podrán introducir fechas del año en curso.
14. (FormatoFechaCeros) Mostrar una fecha con formato dd/mm/aaaa utilizando 0 delante de los días o meses de 1 dígito.
15. (FormatoFechaExtendido) Mostrar una fecha con formato DiaSemana, DiaMes de Mes del Año a las horas:minutos:segundos . Por ejemplo: Miércoles, 9 de Diciembre del 2015 a las 18:45:32
16. (SumarFechaFutura) Suma 10 años, 4 meses y 5 días a la fecha actual.

```

1 Hoy es: dijous, 03 de març del 2022
2 Dentro de 10 años, 4 meses y 5 dias será: dijous, 08 de juliol del 2032

```

17. (RestarFechaPasada) Resta 5 años, 11 meses y 18 días a la fecha actual.
  18. (PagoHorasExtra) Introducir el número de horas trabajadas por un empleado y la fecha en las que las trabaja. Si el día fue sábado o domingo el precio hora trabajada es 20€ en caso contrario 15€. Calcula la cantidad de dinero que habrá que pagar al empleado por las horas trabajadas.
  19. (CalculoNomina) Introducir la fecha inicial y final de una nómina y calcular lo que debe cobrar el empleado sabiendo que cada día trabajado recibe 55 € y tiene una retención del 12% sobre el sueldo.
  20. (ClaseAlumno) Crear una clase Alumno con los atributos codigo, nombre, apellidos, fecha\_nacimiento, calificacion . La fecha de nacimiento deberá introducirse como una fecha. Crear constructor, métodos setter y getter y toString . Crear una instancia con los siguientes valores 1, 'Luis', 'Mas Ros', 05/10/1990 , 7.5 . Mostrar los datos del alumno además de su edad.
- ```

1 Alumno{codigo=1, nombre=Luis, apellidos=Mas Ros, fecha=1990-10-05, calificacion=7.5, edad= 31}

```
21. (PlazoEntrega) Introducir la fecha de entrega de un documento y nos diga si está dentro o fuera de plazo teniendo en cuenta que la fecha de entrega límite es la fecha actual.
 22. (RetencionesTrabajadores) Introducir en un array nombre, apellidos y sueldo de varios trabajadores y la fecha de alta en la empresa. Las fechas deberán introducirse como fechas. Recorrer el array y mostrar para cada trabajador la retención que debe aplicarse sobre el sueldo teniendo en cuenta que los trabajadores incorporados antes de 1980 tienen una retención del 20%, los trabajadores con fecha entre 1980 y 2000 una retención del 15% y los trabajadores con fecha posterior al 2000 la retención que aplicaremos será el 5% del sueldo.
 23. (MayorEdad) Realiza un método estático que dada la fecha de nacimiento de una persona indique si es mayor de edad.
 24. (ClaseConversor) Realiza una clase Conversor que tenga las siguientes características: Toma como parámetro en el constructor un valor entero. Tiene un método getNumero que dependiendo del parámetro devolverá el mismo número (String) en el siguiente sistema de numeración: B Binario, H Hexadecimal, O Octal. Realiza un método main en la clase para probar todo lo anterior.
 25. (ConversorFechas) Realiza una clase ConversorFechas que tenga los siguientes métodos:
 - String normalToAmericano(String fecha) . Este método convierte una fecha en formato normal dd/mm/yyyy a formato americano `mm/dd/yyyy`
 - String americanoToNormal(String fecha) . Este método realiza el paso contrario, convierte fechas en formato americano a formato normal.

9 de diciembre de 2025

19. 6.4 Talleres

19.1. Taller UD05_01: GitHub Classroom

19.1.1. Unirnos a GitHub Classroom

Aceptamos el *Assignement* (la tarea/ejercicio) a partir del link del profesor, en este caso: <https://classroom.github.com/a/qMqg-kz>

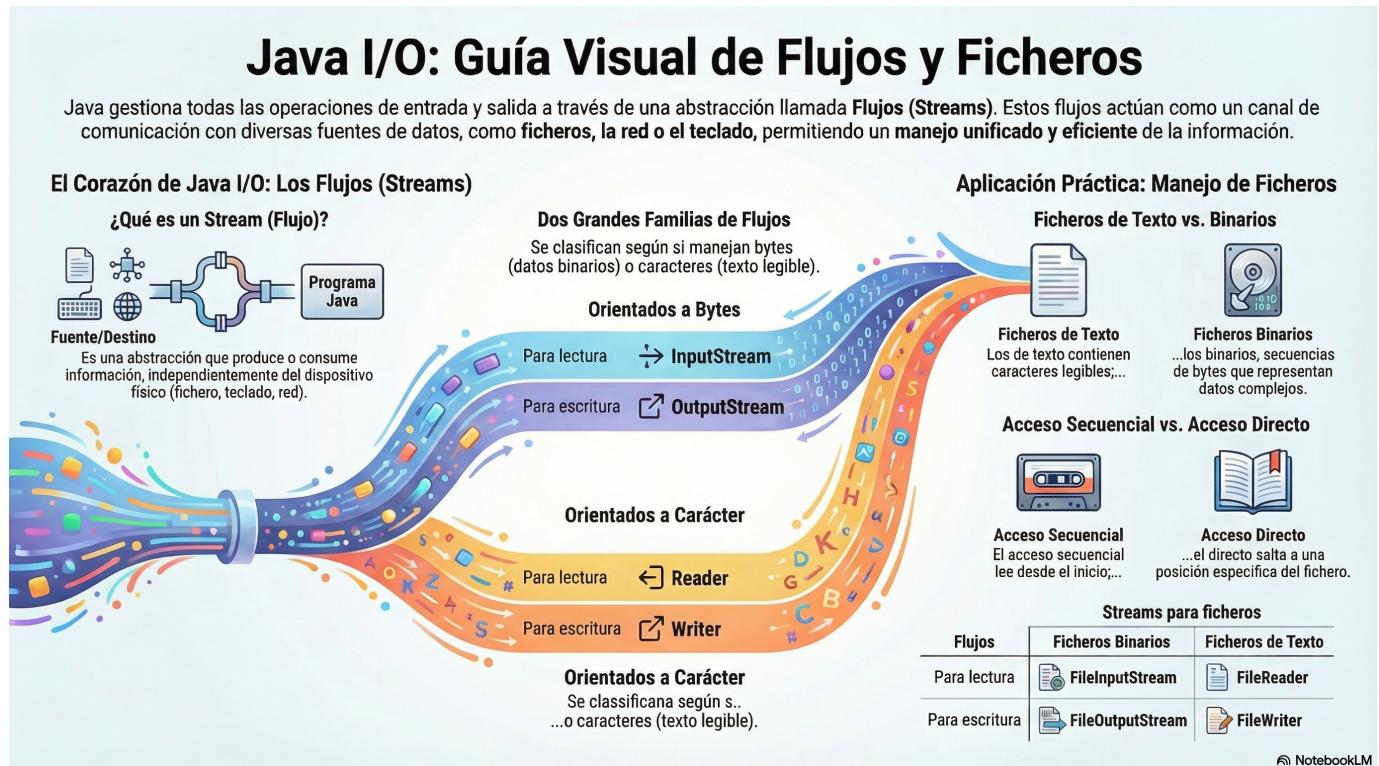
19.1.2. Tarea

Debes enviar tus soluciones a GitHub Classroom y superar al menos la mitad de los tests, cuantos más tests superados, mejor nota tendrás en la tarea.

 13 de diciembre de 2025

7. UD06

20. 7.1 Lectura y escritura de información



En Java, la interacción de nuestro programa con el mundo exterior—desde leer una pulsación de tecla hasta comunicarse a través de una red—se maneja de manera elegante y simplificada a través del concepto de **Streams (flujos)**. Imagina un *stream* como un conducto lógico universal: Java se encarga de la compleja tarea de vincular este conducto a cualquier dispositivo físico (teclado, monitor, ficheros, *sockets*), **liberándote** de tener que codificar para cada dispositivo individualmente. Estos canales se clasifican en flujos orientados a **bytes** (`InputStream` y `OutputStream`), perfectos para datos binarios, y flujos orientados a **caracteres** (`Reader` y `Writer`), esenciales para manejar la codificación Unicode y garantizar que tu aplicación sea internacionalizable. Más allá de la memoria volátil, veremos cómo asegurar que los datos persistan mediante el uso de **ficheros** (de texto o binarios). Finalmente, exploraremos potentes mecanismos como la **serialización**, que permite guardar y transmitir el estado completo de un objeto, y el uso de **sockets** para establecer la comunicación fundamental entre programas cliente y servidor.

20.1. Streams (Flujos)

Los programas Java realizan las operaciones de entrada y salida a través de lo que se denominan Streams (traducido, flujos).



Un stream es una abstracción de todo aquello que produzca o consuma información. Podemos ver a este stream como una entidad lógica que, por otra parte, se encontrará vinculado con un dispositivo físico. La eficacia de esta forma de implementación radica en que las operaciones de entrada y salida que el programador necesita manejar son las mismas independientemente del dispositivo con el que estemos actuando. Será Java quien se encargue de manejar el dispositivo concreto, ya se trate del teclado, el monitor, un sistema de ficheros o un socket de red, etc., liberando a nuestro código de tener que saber con quién está interactuando.

20.1.1. Clasificación de los Streams

En Java los Streams se materializan en un conjunto de clases y subclases, contenidas en el paquete `java.io`. Todas las clases para manejar streams parten, de cuatro clases abstractas:

- `InputStream`
- `OutputStream`
- `Reader`
- `Writer`

Streams	Orientados a bytes	Orientados a carácter
Para lectura	<code>InputStream</code>	<code>Reader</code>
Para escritura	<code>OutputStream</code>	<code>Writer</code>

20.1.1.1. STREAMS ORIENTADOS A BYTE (BYTE STREAMS)

Proporcionan un medio adecuado para el manejo de entradas y salidas de bytes y su uso lógicamente está orientado a la lectura y escritura de datos binarios. El tratamiento del flujo de bytes viene gobernado por dos clases abstractas que son `InputStream` y `OutputStream`.

Cada una de estas clases abstractas tiene varias subclases concretas que controlan las diferencias entre los distintos dispositivos de I/O que se pueden utilizar. Así mismo, estas dos clases son las que definen los métodos que sus subclases tendrán implementados y, de entre todas, destacan las operaciones `read()` y `write()` que leen y escriben bytes de datos respectivamente.

20.1.1.2. STREAMS ORIENTADOS A CARACTER (CHARACTER STREAMS)

Proporciona un medio conveniente para el manejo de entradas y salidas de caracteres. Dichos flujos usan codificación Unicode y, por tanto, se pueden internacionalizar.

Una observación: Este es un modo que Java nos proporciona para manejar caracteres pero al nivel más bajo todas las operaciones de I/O son orientadas a byte. Al igual que la anterior el flujo de caracteres también viene gobernado por dos clases abstractas: `Reader` y `Writer`. Dichas clases manejan flujos de caracteres Unicode. Y también de ellas derivan subclases concretas que implementan los métodos definidos en ellas siendo los más destacados los métodos `read()` y `write()` que, en este caso, leen y escriben caracteres de datos respectivamente.

Como hemos comentado, tanto en `Reader` como en `InputStream` encontramos un método `read()`, en concreto `public int read()`. Observar la diferencia entre estos dos métodos nos ayudará a comprender la diferencia entre los flujos orientados a byte y los orientados a carácter.

método	devuelve
<code>int InputStream.read()</code>	valor entre 0 y 255
<code>int Reader.read()</code>	valor entre 0 y 65535

A pesar de llamarse igual, `InputStream.read()` devuelve el siguiente byte de datos leído del stream. El valor que devuelve está entre 0 y 255 (ó -1 si se ha llegado al final del stream). `Reader.read()`, sin embargo, devuelve un valor entre 0 y 65535 (ó -1), correspondiente al siguiente carácter simple leído del stream.

20.1.2. Stream estándar

Existen una serie de streams de uso común a los cuales se denomina streams estándar. El sistema se encarga de crear estos streams automáticamente.

- `System.in`
- Instancia de la clase `InputStream`: flujo de bytes de entrada
- Métodos:
 - `read()` permite leer un byte de la entrada como entero
 - `skip(n)` ignora `n` bytes de la entrada
 - `available()` número de bytes disponibles para leer en la entrada
- `System.out`
- Instancia de la clase `PrintStream`: flujo de bytes de salida
- Métodos
 - para impresión de datos: `print()`, `println()`
 - `flush()` vacía el buffer de salida escribiendo su contenido
- `System.err`
- Funcionamiento similar a `System.out`
- Se utiliza para enviar mensajes de error (por ejemplo a un fichero de log o a la consola)

Información

Por defecto, `System.in`, `System.out` y `System.err` se encuentran asociados a la consola (teclado y pantalla), pero es posible redirigirlos a otras fuentes o destinos, como por ejemplo a un fichero o una impresora.

20.1.3. Utilización de Streams

Para utilizar un stream hay que seguir una serie de pasos:

- Lectura
- Abrir el stream asociado a una fuente de datos (creación del objeto stream)
- Teclado
- Fichero
- Socket remoto

- Mientras existan datos disponibles
- Leer datos
- Cerrar el stream (método close)
- Escritura
- Abrir el stream asociado a una fuente de datos (creación del objeto stream)
- Pantalla
- Fichero
- Socket local
- Mientras existan datos disponibles
- Escribir datos
- Cerrar el stream (método close)

⚠️ Importante:

- Los Stream estándar ya se encarga el sistema de abrirlos y cerrarlos
- Un fallo en cualquier punto del proceso produce una `IOException`

Consulta los ejemplos [Ejemplo Streams](#)

20.2. Ficheros

En ocasiones necesitamos que los datos que introduce el usuario o que produce un programa persistan cuando éste finaliza, es decir, que se conserven cuando el programa termina su ejecución. Para ello es necesario el uso de una base de datos o de ficheros, que permitan guardar los datos en un almacenamiento secundario como un pendrive, disco duro, DVD, etc. En esta unidad se abordan distintos aspectos relacionados con el almacenamiento en ficheros:

- Introducción a conceptos básicos como los de registro y campo.
- Clasificación de los ficheros según el contenido y forma de acceso.
- Operaciones básicas con ficheros de distinto tipo.

20.2.1. Registros y campos

Llamamos campo a un dato en particular almacenado en una base de datos o en un fichero. Un campo puede ser en nombre de un cliente, la fecha de nacimiento de un alumno, el número de teléfono de un comercio. Los campos pueden ser de distintos tipos: alfanuméricos, numéricos, fechas, etc.

La agrupación de uno o más campos forman un registro. Un registro de alumno podría consistir, por ejemplo, de los siguientes campos

1. Número de expediente.
2. Nombre y apellidos
3. Domicilio
4. Grupo al que pertenece.

Un fichero puede estar formado por registros, lo cual lo dotaría al archivo de estructura. En un fichero de alumnos tendríamos un registro por cada alumno. Los campos del registro serían cada uno de los datos que se almacena del alumno: N° expediente, nombre, etc ...

En Java no existen específicamente los conceptos de campo y registro. Lo más similar que conocemos son las clases (similares a un registro) y, dentro de las clases, los atributos (similares a campos).

Tampoco en Java los ficheros están formados por registros. Java considera los archivos simplemente como flujos secuenciales de bytes. Cuando se abre un fichero se asocia a él un flujo (stream) a través del cual se lee o escribe en el fichero.

Fichero:

```

1 65255
2 José Mateo Ruiz
3 C/ Paz, ...
4 56488
5 Ángela Lopez Villa
6 Av. Blas..
7 24645
8 Armando García Ledesma
9 C/ Tuej ...
10 54654
11 Tomás Ferrando Tamarit
12 C/ Poeta ...

```

```

1 65255;José Mateo Ruiz;C/ Paz, ...
2 56488;Ángela Lopez Villa;Av. Blas..
3 24645;Armando García Ledesma;C/ Tuej ...
4 54654;Tomás Ferrando Tamarit;C/ Poeta ...

```

Registro:

```

1 24645
2 Armando García Ledesma
3 C/ Tuej ...

```

Campo:

```
1 Armando García Ledesma
```

20.2.2. Ficheros de texto vs ficheros binarios.

Desde un punto de vista a muy bajo nivel, un fichero es un conjunto de bits almacenados en memoria secundaria, accesibles a través de una ruta y un nombre de archivo.

Este punto de vista a bajo nivel es demasiado simple, pues cuando se recupera y trata la información que contiene el fichero, esos bits se agrupan en unidades mayores que las dotan de significado. Así, dependiendo de cuál es el contenido del fichero (de cómo se interpretan los bits que contiene el fichero), podemos distinguir dos tipos de ficheros:

- Ficheros de texto (o de caracteres)
- Ficheros binarios (o de bytes)

Un fichero de texto está formado únicamente por caracteres. Los bits que contiene se interpretan atendiendo a una tabla de caracteres, ya sea ASCII o Unicode. Este tipo de ficheros se pueden abrir con un editor de texto plano y son, en general, legibles. Por ejemplo, los ficheros .java que contienen los programas que elaboramos, son ficheros de texto.

Por otro lado, los ficheros binarios contienen secuencias de bytes que se agrupan para representar otro tipo de información: números, sonidos, imágenes, etc. Un fichero binario se puede abrir también con un editor de texto plano pero, en este caso, el contenido será ininteligible. Existen muchos ejemplos de ficheros binarios: el archivo .exe que contiene la versión ejecutable de un programa es un fichero binario.

Las operaciones de lectura/escritura que utilizamos al acceder desde un programa a un fichero de texto están orientadas al carácter: leer o escribir un carácter, una secuencia de caracteres, una línea de texto, etc. En cambio las operaciones de lectura/escritura en ficheros binarios están orientadas a byte: se leen o escriben datos binarios, como enteros, bytes, double, etc.

20.2.3. Acceso secuencial vs acceso directo.

Existen dos maneras de acceder a la información que contiene un fichero:

- Acceso secuencial
- Acceso directo (o aleatorio)

Con acceso secuencial, para poder leer el byte que se encuentra en determinada posición del archivo es necesario leer, previamente, todos los bytes anteriores. Al escribir, los datos se sitúan en el archivo uno a continuación del otro, en el mismo orden en que se introducen. Es decir, la nueva información se coloca en el archivo a continuación de la que ya hay. No es posible realizar modificaciones de los datos existentes, tan solo añadir al final.

Sin embargo, con el acceso directo, es posible acceder a determinada posición (dirección) del fichero de manera directa y, posteriormente, hacer la operación de lectura o escritura deseada.

No siempre es necesario realizar un acceso directo a un archivo. En muchas ocasiones el procesamiento que realizamos de sus datos consiste en la escritura o lectura de todo el archivo siguiendo el orden en que se encuentran. Para ello basta con un acceso secuencial.

20.2.4. Streams para trabajar con ficheros.

Para trabajar con ficheros disponemos de las siguientes clases:

Streams para ficheros	Ficheros binarios	Ficheros de texto
Para lectura	<code>FileInputStream</code>	<code>FileReader</code>
Para escritura	<code>FileOutputStream</code>	<code>FileWriter</code>

- `FileReader` proporciona operaciones para leer de un fichero uno o varios caracteres
- `FileWriter` permite escribir en un fichero uno o varios caracteres o un String.
- `FileInputStream` permite leer bytes de un fichero.
- `FileOutputStream` permite escribir bytes de un fichero.

Información

Consulta en la documentación los distintos constructores disponibles para estas clases.

Observa los ejemplos [P2_1_CrearFichero](#) y [P2_2_SobrescribirFichero](#)

20.2.4.1. LECTURA Y ESCRITURA DE INFORMACIÓN ESTRUCTURADA.

Si observamos la documentación de las clases `FileInputStream` y `FileOutputStream` veremos que las operaciones de lectura y escritura son muy básicas y permiten únicamente leer o escribir uno o varios bytes. Es decir, son operaciones de muy bajo nivel. Si lo que queremos es escribir información binaria más compleja, como por ejemplo un dato de tipo `double` o `boolean` o `int`, tendríamos que hacerlo a través de un stream que permitiese ese tipo de operaciones y asociarlo al `FileOutputStream` o `FileInputStream`.

Podríamos, por ejemplo, asociar un `DataInputStream` a un `FileInputStream` para leer del fichero un dato de tipo `int`.

En ejemplos posteriores se ilustrará cómo asociar un stream a un `FileInputStream`.

Observa los ejemplos [P2_3_LecturaSecuencialTexto](#), [P2_4_EscrituraSecuencialTexto](#), [P2_6_escritura-de-un-fichero-secuencial-binario](#) y [P2_7_LecturaSecuencialBinario](#),

20.2.5. Ficheros con buffering.

Cualquier operación que implique acceder a memoria externa es muy costosa, por lo que es interesante intentar reducir al máximo las operaciones de lectura/escritura que realizamos sobre los ficheros, haciendo que cada operación lea o escriba muchos caracteres. Además, eso también permite operaciones de más alto nivel, como la de leer una línea completa y devolverla en forma de cadena.

Referencia

En el libro Head First Java, describe los buffers de la siguiente forma: "Si no hubiera buffers, sería como comprar sin un carrito: debería llevar los productos uno a uno hasta la caja. Los buffers te dan un lugar en el que dejar temporalmente las cosas hasta que está lleno. Por ello has de hacer menos viajes cuando usas el carrito."

Las clases `BufferedReader`, `BufferedWriter`, `BufferedInputStream` y `BufferedOutputStream` permiten realizar buffering. Situadas "por delante" de un stream de fichero acumulan las operaciones de lectura y escritura y cuando hay suficiente información se llevan finalmente al fichero.

Acción

Recuerda la importancia de cerrar los flujos para asegurarte que se vacía el buffer.

Observa el ejemplo [P2_5_Buffers](#)

20.2.6. Combinación de Streams

En muchas ocasiones, una sola clase de las vistas no nos da la funcionalidad necesaria para poder hacer la tarea que se requiere. En tales casos es necesario combinar varios Streams de manera que unos actúan como origen de información de los otros, o unos escriben sobre los otros.

En este caso tendríamos que combinar tres clases:

20.2.6.1. PARA LA ENTRADA/LECTURA:

```
1 FileInputStream fis = new FileInputStream("fichero.bin");
2 BufferedInputStream bis = new BufferedInputStream(fis);
3 DataInputStream dis = new DataInputStream(bis);
4 //o bien
5 DataInputStream dis = new DataInputStream(new BufferedInputStream(new FileInputStream("fichero.bin")));
```

```
sequenceDiagram
    participant Usuario
    participant FileInputStream
    participant BufferedInputStream
    participant DataInputStream
    participant Stream as "Stream (Fichero Binario)"

    Usuario->>+FileInputStream: Crea FileInputStream (abre el fichero)
    FileInputStream->>+BufferedInputStream: Encadena BufferedInputStream (para mejorar eficiencia)
    BufferedInputStream->>+DataInputStream: Encadena DataInputStream (para leer tipos primitivos)

    DataInputStream->>+Stream: Solicita datos del fichero
    Stream-->DataInputStream: Devuelve datos leídos (bytes)
    DataInputStream-->Usuario: Devuelve datos convertidos (ej: int, double)

    Usuario->>DataInputStream: Cierra DataInputStream
    DataInputStream->>BufferedInputStream: Cierra BufferedInputStream
    BufferedInputStream->>FileInputStream: Cierra FileInputStream
```

20.2.6.2. PARA LA SALIDA/ESCRITURA

```
1 FileOutputStream fos = new FileOutputStream("fichero.bin");
2 BufferedOutputStream bos = new BufferedOutputStream(fos);
3 DataOutputStream dos = new DataOutputStream(bos);
4 //o bien
5 DataInputStream dos = new DataInputStream(new BufferedInputStream(new FileInputStream("fichero.bin")));
```

```
sequenceDiagram
    participant Usuario
    participant FileOutputStream
    participant BufferedOutputStream
    participant DataOutputStream
    participant Stream as "Stream (Fichero Binario)"

    Usuario->>+FileOutputStream: Crea FileOutputStream (abre el fichero)
    FileOutputStream->>+BufferedOutputStream: Encadena BufferedOutputStream (para mejorar eficiencia)
    BufferedOutputStream->>+DataOutputStream: Encadena DataOutputStream (para escribir tipos primitivos)

    Usuario->>DataOutputStream: Escribe datos (ej: writeInt(), writeDouble(), etc.)
    DataOutputStream-->+Stream: Envía datos al fichero binario
    Stream-->DataOutputStream: Confirma escritura

    Usuario->>DataOutputStream: Cierra DataOutputStream
    DataOutputStream->>BufferedOutputStream: Cierra BufferedOutputStream
    BufferedOutputStream->>FileOutputStream: Cierra FileOutputStream
```

20.2.7. try vs try(with resources)

En ocasiones el propio IDE nos sugiere que usemos el bloque `try with resources` en lugar de un simple `try`, así una sentencia como esta:

```
1 FileReader fr = new FileReader(path);
2 BufferedReader br = new BufferedReader(fr);
3 try {
4     return br.readLine();
5 } finally {
6     br.close();
7     fr.close();
8 }
```

Acaba convertida en algo parecido a esta:

```

1 static String readFirstLineFromFile(String path) throws IOException {
2     try (FileReader fr = new FileReader(path);
3          BufferedReader br = new BufferedReader(fr)) {
4         return br.readLine();
5     }
6 }
```

Recomendación

La principal diferencia es que hasta java 7 sólo se podía hacer como en la primera versión. Además en la segunda versión nos "ahorramos" tener que cerrar los recursos, puesto que lo realizará automáticamente en caso de que se produzca algún error evitando así el enmascaramiento de excepciones. Por tanto, sigue siendo necesario cerrar el stream por ejemplo al usar un Buffer para que se vacíe totalmente en el fichero de destino.

20.3. Serialización

Java facilita el almacenamiento y transmisión del estado de un objeto mediante un mecanismo conocido con el nombre de serialización.

La serialización de un objeto consiste en generar una secuencia de bytes lista para su almacenamiento o transmisión. Después, mediante la deserialización, el estado original del objeto se puede reconstruir.

Para que un objeto sea serializable, ha de implementar la interfaz `java.io.Serializable` (que lo único que hace es marcar el objeto como serializable, sin que tengamos que implementar ningún método).

```

1 import java.io.*;
2
3 public class Persona implements Serializable {
4
5     private String nombre;
6     transient private int edad; //No se guardará al serializar
7     private double salario;
8     private Persona tutor;
9     [...]
```

Para que un objeto sea serializable, todas sus variables de instancia han de ser serializables.

Todos los tipos primitivos en Java son serializables por defecto (igual que los arrays y otros muchos tipos estándar).

Cuando queremos evitar que cualquier campo persista en un archivo, lo marcamos como transitorio (`transient`). No podemos marcar ningún método transitorio, solo campos.

Importante

El fichero con los objetos serializados almacena los datos en un formato propio de Java, por lo que no se puede leer fácilmente con un simple editor de texto (ni editar).

Observa el package de ejemplo [UD06.P3_Serializacion](#)

20.4. Sockets

Los sockets son un mecanismo que nos permite establecer un enlace entre dos programas que se ejecutan independientes el uno del otro (generalmente un programa cliente y un programa servidor) Java por medio de la librería `java.net` nos provee dos clases: `Socket` para implementar la conexión desde el lado del cliente y `ServerSocket` que nos permitirá manipular la conexión desde el lado del servidor.

Cabe resaltar que tanto el cliente como el servidor no necesariamente deben estar implementados en Java, solo deben conocer sus direcciones IP y el puerto por el cual se comunicarán.



Observa el package de ejemplo [UD06.P4_Sockets](#)

20.5. Manejo de ficheros y carpetas (File)

La clase `File` es una representación abstracta de ficheros y carpetas. Cuando creamos en Java un objeto de la clase `File` en representación de un fichero o carpeta concretos, no creamos el fichero al que se representa. Es decir, el objeto `File` representa al archivo o carpeta de disco, pero no es el archivo o carpeta de disco.

La clase `File` dispone de métodos que permiten realizar determinadas operaciones sobre los ficheros. Podríamos, por ejemplo, crear un objeto de tipo `File` que represente a `c:\datos\libros.txt` y, a través de ese objeto `File`, realizar consultas relativas al fichero `libros.txt`, como su tamaño, atributos, etc., o realizar operaciones sobre él: borrarlo, renombrarlo, ...

20.5.1. Constructores

La clase `File` tiene varios constructores, que permiten referirse, de varias formas, al archivo que queremos representar:

Método	Descripción
<code>public File (String ruta)</code>	Crea el objeto <code>File</code> a partir de la ruta indicada. Si se trata de un archivo tendrá que indicar la ruta y el nombre.
<code>public File (String ruta, String nombre)</code>	Permite indicar de forma separada la ruta del archivo y su nombre
<code>public File (File ruta, String nombre)</code>	Permite indicar de forma separada la ruta del archivo y su nombre. En este caso la ruta está representada por otro objeto <code>File</code> .
<code>public File (URI uri)</code>	Crea el objeto <code>File</code> a partir de un objeto <code>URI</code> (Uniform Resource Identifier). Un <code>URI</code> permite representar un elemento siguiendo una sintaxis concreta, un estándar.

20.5.2. Métodos

Aquí exponemos algunos métodos interesantes. Hay otros que puedes consultar en la documentación de Java.

Relacionados con el nombre del fichero	
<code>String getName()</code>	Devuelve el nombre del fichero o directorio al que representa el objeto. (Solo el nombre, sin la ruta)
<code>String getPath()</code>	Devuelve la ruta del fichero o directorio. La ruta obtenida es dependiente del sistema, es decir, contendrá el carácter de separación de directorios que esté establecido por defecto. Este separador está definido en <code>public static final String separator</code>
	Devuelve la ruta absoluta del fichero o directorio.

Relacionados con el nombre del fichero

```
String  
getAbsolutePath()
```

`String getParent()` Devuelve la ruta del directorio en que se encuentra el fichero o directorio representado. Devuelve null si no hay directorio padre.

Para hacer comprobaciones

```
boolean exists()  
boolean canWrite()  
boolean canRead()  
boolean isFile()  
boolean  
isDirectory()
```

Permiten averiguar, respectivamente, si el fichero existe, si se puede escribir en él, si se puede leer de él, si se trata de un fichero o si se trata de un directorio

Obtener información de un fichero

```
long length
```

Devuelve el tamaño en bytes del archivo. El resultado es indefinido si se consulta sobre un directorio o una unidad.

```
long  
lastModified
```

Devuelve la fecha de la última modificación del archivo. Devuelve el número de milisegundos transcurridos desde el 1 de enero de 1970

Para trabajar con directorios

```
boolean mkdir()
```

Crea el directorio al cual representa el objeto File.

```
boolean mkdirs()
```

Crea el directorio al cual representa el objeto File, incluyendo todos aquellos que sean necesarios y no existan.

```
String[] list()
```

Devuelve un array de strings con los nombres de los ficheros y directorios que contiene el directorio al que representa el objeto File.

```
String[] list(FileNameFilter  
filtro)
```

Devuelve un array de strings con los nombres de los ficheros y directorios que contiene el directorio al que representa el objeto File y que cumplen con determinado filtro.

```
public File[] listFiles()
```

Devuelve un array de objetos File que representan a los archivos y carpetas contenidos en el directorio al que se refiere el objeto File.

Para hacer cambios

```
boolean renameTo(File  
nuevoNombre)
```

Permite renombrar un archivo. Hay que tener en cuenta que la operación puede fracasar por muchas razones, y que será dependiente del sistema: Que no se pueda mover el fichero de un lugar a otro, que ya exista un fichero que coincide con el nuevo, etc. El método devuelve true solo si la operación se ha realizado con éxito. Existe un método move en la clase Files para mover archivos de una forma independiente del sistema.

```
boolean delete()
```

Elimina el archivo o la carpeta a la que representa el objeto File. Si se trata de una carpeta tendrá que estar vacía. Devuelve true si la operación tiene éxito.

```
boolean createNewFile()
```

Crea un archivo vacío. Devuelve true si la operación se realiza con éxito.

```
File createTempFile(String  
prefijo, String sufijo)
```

Crea un archivo vacío en la carpeta de archivos temporales. El nombre llevará el prefijo y sufijo indicados. Devuelve el objeto File que representa al nuevo archivo.

Observa el ejemplo [UD06.P5_1_Manejo](#)

20.6. Ejemplos UD06

20.6.1. Ejemplo Streams

20.6.1.1. ESTÁNDAR DE ENTRADA

Veamos un ejemplo en el que se lee por teclado hasta pulsar la tecla de retorno, en ese momento el programa acabará imprimiendo por la salida estándar la cadena leída.

Para ir construyendo la cadena con los caracteres leídos podríamos usar la clase `StringBuffer` o la `StringBuilder`. La clase `StringBuffer` permite almacenar cadenas que cambiarán en la ejecución del programa. `StringBuilder` es similar, pero no es síncrona. De este modo, para la mayoría de las aplicaciones, donde se ejecuta un solo hilo, supone una mejora de rendimiento sobre `StringBuffer`.

El proceso de lectura ha de estar en un bloque `try..catch`.

```

1 package UD06.P1_Flujos;
2
3 import java.io.IOException;
4
5 public class P1_1_FlujoEstandarEntrada {
6
7     public static void main(String[] args) {
8         // Cadena donde iremos almacenando los caracteres que se escriban
9         StringBuilder str = new StringBuilder();
10        char c;
11        // Por si ocurre una excepción ponemos el bloque try-cath
12        try {
13            // Mientras la entrada de teclado no sea Intro
14            while ((c = (char) System.in.read()) != '\n') {
15                // Añadir el carácter leído a la cadena str
16                str.append(c);
17            }
18        } catch (IOException ex) {
19            System.out.println(ex.getMessage());
20        }
21        // Escribir la cadena que se ha ido tecleando
22        System.out.println("Cadena introducida: " + str);
23    }
24 }
```

20.6.1.2. ESTÁNDAR DE SALIDA

```

1 package UD06.P1_Flujos;
2
3 import java.io.BufferedReader;
4 import java.io.FileWriter;
5 import java.io.IOException;
6 import java.io.InputStreamReader;
7 import java.io.PrintWriter;
8
9 public class P1_2_FlujoEstandarSalida {
10
11     public static void main(String[] args) {
12         // Por si ocurre una excepción ponemos el bloque try-cath
13         try {
14             PrintWriter out = null;
15             out = new PrintWriter(new FileWriter("test/salida.txt", true));
16             BufferedReader br = new BufferedReader(
17                 new InputStreamReader(System.in));
18             String s;
19             while (!(s = br.readLine()).equals("salir")) {
20                 out.println(s);
21             }
22             out.close();
23         } catch (IOException ex) {
24             System.out.println(ex.getMessage());
25         }
26     }
27 }
```

20.6.2. Ficheros

20.6.2.1. CREAR UN FICHERO

En el siguiente ejemplo vemos como crear un fichero de texto y escribir una frase en él.

```

1 package UD06.P2_Ficheros;
2
3 import java.io.*;
4
5 public class P2_1_CrearFichero {
6
7     public static void main(String[] args) {
8         FileWriter f = null;
9         try {
10             f = new FileWriter("texto.txt");
11             f.write("Este texto se escribe en el fichero\n\r");
12         } catch (IOException e) {
13             System.out.println("Problema al abrir o escribir ");
14         } finally {
15             if (f != null) {
16                 try {
17                     f.close();
18                 } catch (IOException e) {
19                     System.out.println("Problema al cerrar el fichero");
20                 }
21             }
22         }
23     }
24 }
```

La creación del `FileWriter` puede provocar `IOException`, lo mismo que el método `write`. Por ello las instrucciones se encuentran en un bloque `try-catch`.

Al finalizar su uso, y tan pronto como sea posible, hay que cerrar los streams (`close`) .

20.6.2.2. SOBRESCRIBIR UN FICHERO

Es muy importante tener en cuenta que cuando se crea un `FileWriter` o un `FileOutputStream` y se escribe en él ...

- ... si el fichero no existe se crea
- ... si el fichero existe, **su contenido se reemplaza** por el nuevo. El contenido previo que tuviera el fichero se pierde.

Es posible escribir en un fichero indicando que la información se añada a la que ya hay y no se reescriba el fichero. Para ello usaremos el constructor que recibe el parámetro `append` y lo pasaremos a true.

El siguiente ejemplo muestra como añadir una línea al final de un fichero de texto.

```

1 package UD06.P2_Ficheros;
2
3 import java.io.*;
4
5 public class P2_2_SobrescribirFichero {
6
7     public static void main(String[] args) {
8         try (FileWriter f = new FileWriter("david/enero/texto.txt", true)) {
9             f.write("Este texto se añade en el fichero\n\r");
10        } catch (IOException e) {
11            System.out.println("Problema al abrir o escribir ");
12        }
13    }
14 }
```

En este ejemplo se ha utilizado la nueva sintaxis disponible para los bloques `try-catch`: lo que se denomina "try with resources". Esta sintaxis permite crear un objeto en la cabecera del bloque `try`. El objeto creado se cerrará automáticamente al finalizar. El objeto debe pertenecer al interface `Closeable`, es decir, debe tener método `close()`.

20.6.2.3. OPERACIONES CON FICHEROS DE ACCESO SECUENCIAL

Como hemos comentado anteriormente el acceso secuencial a un fichero supone que para acceder a un byte es necesario leer previamente los anteriores. Suele utilizarse este tipo de acceso cuando es necesario leer un archivo de principio a fin.

Vamos a ver una serie de ejemplos que muestren cómo leer y escribir secuencialmente un fichero.

20.6.2.3.1. Lectura de un fichero secuencial de texto

Leer un fichero de texto y mostrar el número de vocales que contiene.

```

1 package UD06.P2_Ficheros;
2
3 import java.io.*;
4
5 public class P2_3_LecturaSecuencialTexto {
6
7     final static String VOCALES = "AEIOUaeiou";
8
9     public static void main(String[] args) {
10        try (FileReader f = new FileReader(new File("texto.txt")));) {
11            int contadorVocales = 0;
12            int caracter;
13            while ((caracter = f.read()) != -1) {
14                char letra = (char) caracter;
15                if (VOCALES.indexOf(letra) != -1) {
16                    contadorVocales++;
17                }
18            }
19            System.out.println("Numero de vocales: " + contadorVocales);
20        } catch (FileNotFoundException e) {
21            System.out.println("Problema al abrir el fichero");
22        } catch (IOException e) {
23            System.out.println("Problema al leer");
24        }
25    }
26 }
```

Observa que:

- Para leer el fichero de texto usamos un `InputStream`.
- Al crear el stream (`InputStream`) es posible indicar un objeto de tipo `File`
- La operación `read()` devuelve un entero. Para obtener el carácter correspondiente tenemos que hacer una conversión explícita de tipos.
- La operación `read()` devuelve -1 cuando no queda información que leer del stream.
- La guarda del bucle `while` combina una asignación con una comparación. En primer lugar se realiza la asignación y luego se compara carácter con -1.
- `FileNotFoundException` sucede cuando el fichero no se puede abrir (no existe, permiso denegado, etc), mientras que `IOException` se lanzará si falla la operación `read()`

20.6.2.3.2. Escritura de un fichero secuencial de texto

Dada una cadena escribirla en un fichero en orden inverso:

```

1 package UD06.P2_Ficheros;
2
3 import java.io.*;
4
5 public class P2_4_EscrituraSecuencialTexto {
6
7     final static String CADENA = "En un lugar de la mancha...";
8
9     public static void main(String[] args) {
10        try (FileWriter f = new FileWriter(new File("texto.txt")));) {
11            for (int i = CADENA.length() - 1; i >= 0; i--) {
12                f.write(CADENA.charAt(i));
13            }
14            System.out.println("FIN");
15        } catch (FileNotFoundException e) {
16            System.out.println("Problema al abrir el fichero");
17        } catch (IOException e) {
18            System.out.println("Problema al escribir");
19        }
20    }
21 }
22 }
```

Observa que:

- Para escribir el fichero de texto usamos un `FileWriter`.
- Tal y como se ha creado el stream, el fichero (si ya existe) se sobreescibirá.
- El manejo de excepciones es como el del caso previo.

20.6.2.4. USANDO BUFFERS PARA LEER Y ESCRIBIR DE/EN FICHERO

En el siguiente código se usan buffers para leer líneas de un fichero y escribirlas en otro convertidas a mayúsculas

```

1 package UD06.P2_Ficheros;
2
3 import java.io.*;
4
5 public class P2_5_Buffers {
6
7     final static String ENTRADA = "texto.txt";
8     final static String SALIDA = "textoMayusculas.txt";
9
10    public static void main(String[] args) {
11        try (BufferedReader fe = new BufferedReader(new FileReader(ENTRADA));
12             BufferedWriter fs = new BufferedWriter(new FileWriter(SALIDA))){
13            String linea;
14            while ((linea = fe.readLine()) != null) {
15                fs.write(linea.toUpperCase());
16                fs.newLine();
17            }
18            System.out.println("FIN");
19        } catch (FileNotFoundException e) {
20            System.out.println("Problema al abrir el fichero");
21        } catch (IOException e) {
22            System.out.println("Problema al leer o escribir");
23        }
24    }
25 }
```

Observa que:

- Usamos buffers tanto para leer como para escribir. Esto permite minimizar los accesos a disco.
- Los buffers quedan asociados a un `FileReader` y `FileWriter` respectivamente. Realizamos las operaciones de lectura/escritura sobre las clases `Buffered...` y cuando es necesario la clase accede internamente al stream que maneja el fichero.
- Es necesario escribir explícitamente los saltos de línea. Esto se hace mediante el método `newLine()`. `newLine()` permite añadir un salto de línea sin preocuparnos de cuál es el carácter de salto de línea. El salto de línea es distinto en distintos sistemas: en unos es `\n`, en otros `\r`, en otros `\n\r`, ...
- `BufferedReader` dispone de un método para leer líneas completas (`readLine()`). Cuando se llega al final del fichero este método devuelve `null`.
- Fíjate como el bloque `try with resources` creamos varios objetos. Si la creación de cualquiera de ellos falla, se cerrarán todos los stream que se han abierto.

20.6.2.5. FICHEROS BINARIOS

20.6.2.5.1. Escritura de un fichero secuencial binario

Ya hemos visto que con `FileInputStream` y `FileOutputStream` se puede leer y escribir bytes de información de/a un archivo.

Sin embargo esto puede no ser suficiente cuando la información que tenemos que leer o escribir es más compleja y los bytes se agrupan para representar distintos tipos de datos.

Imaginemos por ejemplo que queremos guardar en un fichero “jugadores.dat”, el año de nacimiento y la estatura de cinco jugadores de baloncesto:

```

1 package UD06.P2_Ficheros;
2
3 import java.io.*;
4 import java.util.Scanner;
5
6 public class P2_6_EscrituraSecuencialBinario {
7
8     public static void main(String[] args) {
9         Scanner tec = new Scanner(System.in);
10        try (DataOutputStream fs = new DataOutputStream(
11                new BufferedOutputStream(
12                    new FileOutputStream("jugadores.dat")))) {
13            for (int i = 1; i <= 5; i++) {
14                //Pedimos datos al usuario
15                System.out.println(" ---- Jugador " + i + " ----");
16                System.out.print("Nombre: ");
17                String nombre = tec.nextLine();
18
19                System.out.print("Nacimiento: ");
20                int anyo = tec.nextInt();
21
22                System.out.print("Estatua: ");
23                double est = tec.nextDouble();
24                //Vaciar salto linea
25                tec.nextLine();
26
27                //Volcamos informacion al fichero
28                fs.writeUTF(nombre);
29                fs.writeInt(anyo);
30                fs.writeDouble(est);
31            }
32        } catch (FileNotFoundException e) {
33            System.out.println("Problema al abrir el fichero");
34        } catch (IOException e) {
35            System.out.println("Problema al leer o escribir");
36        }
37    }
38 }

```

Observa que:

- Para escribir información binaria usamos un `DataInputStream` asociado al stream. La clase tiene métodos para escribir `int`, `byte`, `double`, `boolean`, etc.
- Además, como hemos hecho en ejemplos previos, usamos un buffer. Fíjate como en el constructor se enlazan unas clases con otras.
- A pesar de que en Java los ficheros son secuencias de bytes, estamos dotando al fichero de cierta estructura: primero aparece el nombre, luego el año y finalmente la estatura. Cada uno de estos tres datos constituirían un registro de formado por tres campos. Para poder recuperar información de un fichero binario es necesario conocer cómo se estructura ésta dentro del fichero.

20.6.2.5.2. Lectura de un fichero secuencial binario

```

1 package UD06.P2_Ficheros;
2
3 import java.io.*;
4 import java.util.Scanner;
5
6 public class P2_7_LecturaSecuencialBinario {
7
8     public static void main(String[] args) {
9         Scanner tec = new Scanner(System.in);
10        try (DataInputStream fe = new DataInputStream(new BufferedInputStream(
11                new FileInputStream("jugadores.dat")))) {
12            while (true) {
13                //Leemos nombre
14                System.out.println(fe.readUTF());
15                //leemos y desechamos resto de datos
16                fe.readInt();
17                fe.readDouble();
18            }
19        } catch (EOFException e) {
20            //Se lanzará cuando se llegue al final del fichero
21        } catch (FileNotFoundException e) {
22            System.out.println("Problema al abrir el fichero");
23        } catch (IOException e) {
24            System.out.println("Problema al leer o escribir");
25        }
26    }
27 }

```

Observa que:

- A pesar de que necesitamos solamente el nombre de cada jugador, es necesario leer también el año y la estatura. No es posible acceder al nombre del segundo jugador sin leer previamente todos los datos del primer jugador.

- La lectura se hace a través de un bucle infinito (`while (true)`), que finalizará cuando se llegue el final del fichero y al leer de nuevo se produzca la excepción `EOFException`

20.6.3. Ejemplo de Serialización

En el siguiente ejemplo usaremos una clase persona que definiremos de la siguiente manera

20.6.3.1. PERSONA

```

1 package UD06.P3_Serializacion;
2
3 import java.io.*;
4
5 public class Persona implements Serializable {
6
7     private String nombre;
8     transient private int edad; //No se guardará al serializar
9     private double salario;
10    private Persona tutor;
11
12    public Persona(String nom, double salari) {
13        this.nombre = nom;
14        this.salario = salari;
15        edad = 0;
16        tutor = null;
17    }
18
19    public String getNombre() {
20        return nombre;
21    }
22
23    public int getEdad() {
24        return edad;
25    }
26
27    public double getSalario() {
28        return salario;
29    }
30
31    public Persona getTutor() {
32        return tutor;
33    }
34
35    public void incrementaEdad() {
36        edad++;
37    }
38
39    public void asignaTutor(Persona p) {
40        tutor = p;
41    }
42 }
```

Ahora detallamos la clase para serializar o guardar la información en un archivo:

```

1 package UD06.P3_Serializacion;
2
3 import java.io.*;
4
5 public class Guardar {
6
7     public static void main(String args[]) {
8         ObjectOutputStream salida;
9         Persona p1, p2, p3, p4;
10
11         p1 = new Persona("Vicent", 1200.0);
12         p2 = new Persona("Mireia", 1800.0);
13         p3 = new Persona("Josep", 2100.0);
14         p4 = new Persona("Marta", 850.0);
15
16         p1.asignaTutor(p2);
17         p2.asignaTutor(p3);
18         p3.asignaTutor(p4);
19
20         try {
21             salida = new ObjectOutputStream(new FileOutputStream("empleats.ser"));
22             salida.writeObject(p1);
23             salida.close();
24         } catch (IOException e) {
25             System.out.println("Algún problema guardando a disco.");
26         }
27     }
28 }
```

Y por último la clase para Leer la información una vez guardada:

```

1 package UD06.P3_Serializacion;
2
3 import java.io.*;
4
5 public class Leer {
6
7     public static void main(String args[]) {
8         ObjectInputStream entrada;
9         Persona p1, p2, p3, p4;
10
11     try {
12         entrada = new ObjectInputStream(new FileInputStream("empleats.ser"));
13         p1 = (Persona) entrada.readObject();
14         entrada.close();
15
16         p2 = p1.getTutor();
17         p3 = p2.getTutor();
18         p4 = p3.getTutor();
19
20         System.out.println(p4.getNombre());
21         System.out.println(p4.getEdad());
22         System.out.println(p4.getSalario());
23
24     } catch (ClassNotFoundException e) {
25         System.out.println("Algun problema con las clases definidas.");
26     } catch (IOException e) {
27         System.out.println("Algun problema leyendo de disco.");
28     }
29 }
30 }
```

20.6.4. Ejemplo de Sockets

Para nuestro ejemplo de sockets implementaremos ambos (cliente y servidor) usando Java y se comunicarán usando el puerto 11000 (es bueno elegir los puertos en el rango de 1024 hasta 65535).

La secuencia de eventos en nuestro ejemplo será:

- El servidor creará el socket y esperará a que el cliente se conecte o lo detengamos.
- Por otro lado, el cliente abrirá la conexión con el servidor y le enviará una frase en minúsculas que escribirá el usuario y la enviará al servidor.
- Una vez recibida la frase en minúsculas, el servidor la convertirá en mayúsculas, la devolverá al cliente.
- El cliente mostrará la frase en mayúsculas recibida desde el servidor, e irá añadiendo el texto recibido acumulando las frases enviadas.
- Cuando en un envío reciba la palabra EXIT (no importa las mayúsculas) devolverá frase en mayúsculas y cerrará la conexión.
- Si la palabra recibida no es EXIT, el servidor quedará a la espera de una nueva conexión de otro cliente.

20.6.4.1. SERVIDORSOCKET

```

1 import java.io.*;
2 import java.net.*;
3 import java.util.Enumeration;
4
5 public class ServidorSocket {
6
7     private static final int PORT=11000;
8
9     private static void mostrarIPs(StringBuilder sb) {
10        try {
11            Enumeration Interfaces = NetworkInterface.getNetworkInterfaces();
12            while (Interfaces.hasMoreElements()) {
13                NetworkInterface Interface = (NetworkInterface) Interfaces.nextElement();
14                Enumeration Addresses = Interface.getInetAddresses();
15                while (Addresses.hasMoreElements()) {
16                    InetAddress Address = (InetAddress) Addresses.nextElement();
17                    sb.append("\n\t").append(Address.getHostAddress()).append(":").append(PORT);
18                }
19            }
20        } catch (SocketException ex) {
21            System.err.println("Error. Al intentar obtener las interfaces de red.");
22        }
23    }
24
25    public static <serverSocket> void main(String[] args) throws IOException, ClassNotFoundException {
26        String FraseClient;
27        String FraseMajuscules;
28        Socket clientSocket;
29        ObjectInputStream entrada;
30        ObjectOutputStream eixida;
31
32        try (ServerSocket serverSocket = new ServerSocket(PORT)) {
33
34            StringBuilder sb = new StringBuilder();
35            sb.append("Server iniciado y escuchando en la ip y puerto: ");
36            mostrarIPs(sb);
37            System.out.println(sb.toString());
38
39            while (true) {
40                clientSocket = serverSocket.accept();
41                entrada = new ObjectInputStream(clientSocket.getInputStream());
42                FraseClient = (String) entrada.readObject();
43
44                System.out.println("La frase recibida es: " + FraseClient);
45
46                eixida = new ObjectOutputStream(clientSocket.getOutputStream());
47                FraseMajuscules = FraseClient.toUpperCase();
48
49                System.out.println("El server devuelve la frase: " + FraseMajuscules);
50
51                eixida.writeObject(FraseMajuscules);
52                clientSocket.close();
53
54                System.out.println("Server esperando una nueva conexión...");
55            }
56        } catch (ClassNotFoundException ex) {
57            System.err.println("Error. Clase no encontrada");
58        } catch (IOException ex) {
59            System.err.println("Error. De entrada salida." + ex.toString());
60        }
61    }
62 }

```

20.6.4.2. CLIENTESOCKET

```

1 import java.io.*;
2 import java.net.*;
3 import java.util.Scanner;
4
5 public class ClienteSocket{
6
7     private static final int PORT = 11000;
8
9     //private static final String DNSAWS = "ec2-44-200-177-74.compute-1.amazonaws.com";
10    private static final String DNSAWS = "127.0.0.1";
11
12    public static <string> void main(String[] args) throws IOException, ClassNotFoundException {
13        ObjectInputStream entrada;
14        ObjectOutputStream eixida;
15        String frase;
16        StringBuilder fraseMayuscula = new StringBuilder();
17        String aux = "";
18        int iter = 0;
19
20        do {
21            try (Socket socket = new Socket(DNSAWS, PORT)) {
22                eixida = new ObjectOutputStream(socket.getOutputStream());
23
24                if (iter == 0) {
25                    System.out.println("Introduce la frase a enviar en minúsculas");
26                    iter++;
27                } else {
28                    System.out.println("Muy bonito, ahora introduce otra o \"exit\"/\"EXIT\" para salir");
29                }
30
31                Scanner in = new Scanner(System.in);
32
33                frase = in.nextLine();
34
35                System.out.println("Se envia la frase " + frase);
36
37                eixida.writeObject(frase);
38                entrada = new ObjectInputStream(socket.getInputStream());
39                aux = (String) entrada.readObject().toString();
40
41                if (fraseMayuscula.toString().isBlank()) {
42                    fraseMayuscula.append(aux);
43                } else {
44                    fraseMayuscula.append(" ").append(aux);
45                }
46
47                System.out.println("La frase recibida es: " + fraseMayuscula);
48
49            } catch (Exception ex){
50                System.err.println("Error:" + ex.toString());
51                System.exit (-1);
52            }
53        } while (!aux.equals("EXIT"));
54
55    }
56 }
```

20.6.5. Ejemplo de manejo de ficheros y carpetas

Veamos ahora un ejemplo para mostrar información y contenido de una carpeta:

```

1 package UD06.P5_Manejo;
2
3 import java.io.*;
4 import java.util.*;
5
6 public class P5_1_Manejo {
7
8     public static void main(String[] args) {
9
10        Scanner tec = new Scanner(System.in);
11        System.out.println("Introduce ruta absoluta de una carpeta");
12        String nombreCarpeta = tec.nextLine();
13        //Creamos objeto File para representar a la carpeta
14        File car = new File(nombreCarpeta);
15        //Comprobamos si existe
16        if (car.exists()) {
17            //¿Es una carpeta?
18            if (car.isDirectory()) {
19                if (car.canRead()) {
20                    System.out.println("Lectura permitida");
21                } else {
22                    System.out.println("Lectura no permitida");
23                }
24
25                if (car.canWrite()) {
26                    System.out.println("Escritura permitida");
27                } else {
28                    System.out.println("Escritura no permitida");
29                }
30
31                if (car.isHidden()) {
32                    System.out.println("Carpeta oculta");
33                } else {
34                    System.out.println("Carpeta visible");
35                }
36
37                System.out.println("---- Contenido de la carpeta ----");
38                File[] contenido = car.listFiles();
39                for (File f : contenido) {
40                    System.out.println(f.getName());
41                }
42            } else {
43                System.out.println(car.getAbsolutePath() + " No es una carpeta");
44            }
45        } else {
46            System.out.println(
47                "No existe la carpeta/archivo " + car.getAbsolutePath());
48        }
49    }
50 }
```

20.7. Píldoras informáticas relacionadas

 15 de enero de 2026

21. 7.2 Comparativa CRUD

Tenemos un array: **lista**

y una variable con la ocupación: **ocupacion** (empieza en cero y se va incrementando)

y otra variable **MAX_DISPONIBLE**: con el total máximo admitido.

Métodos	Con huecos	Sin huecos No ordenado	Sin huecos Ordenado
C añadir (create) 	<pre>Si (ocupacion < MAX_DISPONIBLE) recorro toda la lista si encuentro un hueco(null) inserto el elemento y ocupacion++</pre> <p>eficiente facilImplementar </p>	<pre>Si (ocupacion < MAX_DISPONIBLE) inserto el elemento en lista[ocupacion++]</pre> <p>eficiente facilImplementar </p>	<pre>Si (ocupacion < MAX_DISPONIBLE) insertoOrdenado el elemento en lista y ocupacion++</pre> <p>eficiente facilImplementar </p>
R leer (read) 	<pre>Si (ocupacion >0) Recorro toda la lista si encuentro el elemento (equals) lo devuelvo si no devuelvo una excepción de NoEncontrado</pre> <p>eficiente facilImplementar </p>	<pre>Si (ocupacion >0) Recorro la lista hasta ocupacion-1 si encuentro el elemento (equals) lo devuelvo si no devuelvo una excepción de NoEncontrado</pre> <p>eficiente facilImplementar </p>	<pre>Si (ocupacion >0) Usar un metodo de buscarOrdenado devolver el elemento o la Excepcion si no existe</pre> <p>eficiente facilImplementar </p>
U modificar (update) 	<pre>Si (ocupacion >0) Recorro toda la lista si encuentro el elemento (equals) lo modifco devuelvo una excepción de NoEncontrado</pre> <p>eficiente facilImplementar </p>	<pre>Si (ocupacion >0) Recorro la lista hasta ocupacion -1 si encuentro el elemento (equals) lo modifco devuelvo una excepción de NoEncontrado</pre> <p>eficiente facilImplementar </p>	<pre>Si (ocupacion >0) Usar un metodo de buscarOrdenado modificar el elemento o la Excepcion si no existe</pre> <p>eficiente facilImplementar </p>
D borrar (delete) 	<pre>Si (ocupacion >0) Recorro toda la lista si encuentro el elemento (equals) lo borro y ocupacion-- devuelvo una excepción de NoEncontrado</pre> <p>eficiente facilImplementar </p>	<pre>Si (ocupacion >0) Recorro la lista hasta ocupacion-1 si encuentro el elemento (equals) lo borro ocupacion-- //desplazo la lista hacia la izquierda recorro(i) des la posicion actual hasta ocupacion -1 lista[i] = lista[i+1] sino devuelvo una excepción de NoEncontrado</pre> <p>eficiente facilImplementar </p>	<pre>Si (ocupacion >0) Usar un metodo de buscarOrdenado si encuentro el elemento lo borro ocupacion- //desplazo la lista hacia la izquierda sino devuelvo una excepción de NoEncontrado</pre> <p>eficiente facilImplementar </p>

22. 7.3 Ejercicios de la UD06

22.1. Paquetes Completos

22.1.1. Paquete: UD06._1.gestorVuelos

Se desea realizar una aplicación `GestorVuelos` para gestionar la reserva y cancelación de vuelos en una agencia de viajes. Dicha agencia trabaja únicamente con la compañía aérea Iberia, que ofrece vuelos desde/hacia varias ciudades de Europa. Se deben definir las clases que siguen, teniendo en cuenta que sus atributos serán privados y sus métodos sólo los que se indican en cada clase.

1. Implementación de la clase `Vuelo`, que permite representar un vuelo mediante los atributos:

- `identificador (String)`
- `origen (String)`
- `destino (String)`
- `hSalida` (un tipo que te permita controlar la hora, no es un `String` ni un `int`, etc.)
- `hLlegada` (un tipo que te permita controlar la hora, no es un `String` ni un `int`, etc.)
- Además, cada vuelo dispone de 50 asientos, es decir, pueden viajar, como mucho, 50 pasajeros en cada vuelo. Para representarlos, se hará uso de `asiento`, un array de `String` (nombres de los pasajeros) junto con un atributo `numP` que indique el número actual de asientos reservados. Si el asiento `i` está reservado, `asiento[i]` contendrá el nombre del pasajero que lo ha reservado. Si no lo está, `asiento[i]` será `null`. En el array `asiento`, las posiciones impares pertenecen a asientos de ventanilla y las posiciones pares, a asientos de pasillo (la posición 0 no se utilizará).

En esta clase, se deben implementar los siguientes métodos:

- `public Vuelo(String id, String orig, String dest, LocalTime hsal, LocalTime hlleg)`. **Constructor** que crea un vuelo con identificador, ciudad de origen, ciudad de destino, hora de salida y hora de llegada indicados en los respectivos parámetros, y sin pasajeros.
- `public String getIdenficator()`. Devuelve el `identificador`
- `public String getOrigen()`. Devuelve `origen`.
- `public String getDestino()`. Devuelve `destino`.
- `public boolean hayLibres()`. Devuelve `true` si quedan asientos libres y `false` si no quedan.
- `public boolean equals(Object o)`. Dos vuelos son iguales si tienen el mismo identificador.
- `public int reservarAsiento(String pas, char pref) throws VueloCompletoException`. Si el vuelo ya está completo se lanza una excepción. Si no está completo, se reserva al pasajero `pas` el primer asiento libre en `pref`. El carácter `pref` será 'P' o 'V' en función de que el pasajero desee un asiento de pasillo o de ventanilla. En caso de que no quede ningún asiento libre en la preferencia indicada (`pref`), se reservará el primer asiento libre de la otra preferencia. El método devolverá el número de asiento que se le ha reservado. Este método hace uso del método privado `asientoLibre`, que se explica a continuación.
- `private int asientoLibre(char pref)`. Dado un tipo de asiento `pref` (pasillo 'P' o ventanilla 'V'), devuelve el primer asiento libre (el de menor numero) que encuentre de ese tipo. O devuelve 0 si no quedan asientos libres de tipo `pref`.
- `public void cancelarReserva(int numasiento)`. Se cancela la reserva del asiento `numasiento`.
- `public String toString()`. Devuelve una `String` con los datos del vuelo y los nombres de los pasajeros, con el siguiente formato:

```

1 IB101 Valencia París 19:05:00 21:00:00
2 Pasajeros:
3 Asiento 1: Sonia Dominguez
4 ...
5 Asiento 23: Fernando Romero

```

2. Diseñar e implementar una clase Java `TestVuelo` que permita probar la clase `Vuelo` y sus métodos. Para ello se desarrollará el método `main` en el que:

- Se cree el vuelo IB101 de Valencia a París, que sale a las 19:05 y llega a las 21:00
- Reservar:
 - Un asiento de ventanilla a "Miguel Fernández"
 - Un asiento de ventanilla a "Ana Folgado"
 - Un asiento de pasillo a "David Más"
- Mostrar el vuelo por pantalla
- Cancelar la reserva del asiento que indique el usuario.

- Mostrar de nuevo el vuelo por pantalla
3. Implementación de la clase `Compania` para representar todos los vuelos de una compañía aérea. Una Compañía tiene un nombre y puede ofrecer, como mucho, 10 vuelos distintos. Para representarlos se utilizará `listaVuelos`, un array de objetos `Vuelo` junto con un atributo `numVuelos` que indique el número de vuelos que la compañía ofrece en un momento dado. Las operaciones de esta clase son:
- `public Compania(String n) throws FileNotFoundException`. Constructor de una compañía de nombre `n`. Cuando se crea una compañía, se invoca al método privado `leeVuelos()` para cargar la información de vuelos desde un fichero. Si el fichero no existe, se propaga la excepción `FileNotFoundException`
 - `private void leeVuelos() throws FileNotFoundException`. Lee desde un fichero toda la información de los vuelos que ofrece la compañía y los va almacenando en el array de vuelos `listaVuelos`. El nombre del fichero coincide con el nombre de la compañía y tiene extensión `.txt`. La información de cada vuelo se estructura en el fichero como sigue:

```

1 <Identificador>
2 <Origen>
3 <Destino>
4 <Hora de salida>
5 <Minuto de salida>
6 <Hora de llegada>
7 <Minuto de llegada>
8 ...
9 ...

```

Si el fichero no existe, se propaga la excepción `FileNotFoundException`.

- `public Vuelo buscarVuelo(String id) throws ElementoNoEncontradoException`. Dado un identificador de vuelo `id`, busca dicho vuelo en el array de vuelos `listaVuelos`. Si lo encuentra, lo devuelve. Si no, lanza `ElementoNoEncontradoException`.
- `public void mostrarVuelosIncompletos(String o, String d)`. Muestra por pantalla los vuelos con origen `o` y destino `d`, y que tengan asientos libres. Por ejemplo, vuelos con asientos libres de la compañía Iberia con origen Milán y destino Valencia:

```

1 Iberia IB201 Milán Valencia 14:25:00 16:20:00
2 Iberia IB202 Milán Valencia 21:40:00 23:35:00

```

4. En la clase `GestorVuelos` se probará el comportamiento de las clases anteriores. En esta clase se debe implementar el método `main` en el que, por simplificar, se pide únicamente:

- la creación de la compañía aérea `Iberia`. Se dispone de un fichero de texto "`Iberia.txt`", con la información de los vuelos que ofrece.
- Reserva de un asiento de ventanilla en un vuelo de Valencia a París por parte de Manuel Soler Roca. Para ello:
- Mostraremos vuelos con origen Valencia y destino París, que no estén completos.
- Pediremos al usuario el identificador del vuelo en que quiere hacer la reserva.
- Buscaremos el vuelo que tiene el identificador indicado. Si existe realizaremos la reserva y mostraremos un mensaje por pantalla. En caso contrario mostraremos un mensaje de error por pantalla.

22.1.2. Paquete: UD06._2.maquinaExpededora

Se desea simular el funcionamiento de una máquina expendedora. Se trata de una expendedora sencilla que, por el momento, será capaz de dispensar únicamente un producto.

Su funcionamiento, a grandes rasgos, es el siguiente:

1. El cliente introduce dinero en la máquina. Al dinero introducido lo llamaremos `credito`.
2. Selecciona el producto que quiere comprar (ya hemos comentado que por el momento habrá un solo producto).
3. Si hay stock del producto seleccionado, la máquina dispensa el artículo elegido y devuelve el importe sobrante (diferencia entre el crédito introducido y el precio del producto).

Durante el proceso se pueden producir diversas incidencias, como por ejemplo, que el cliente no haya introducido suficiente crédito para comprar el producto, que no quede producto o que no haya cambio suficiente para la devolución. La máquina también da la posibilidad de solicitar la devolución del crédito sin realizar la compra.

1. Diseñar la clase `Expededora` (proyecto `Expededora`) con los atributos y métodos que se describen a continuación.

- Atributos (privados)
- `credito`: Cantidad de dinero (en euros) introducida por el cliente.
- `stock`: Número de unidades que quedan en la máquina disponibles para la venta. Se reducirá con cada nueva venta.

- `precio` : Precio del único artículo que dispensa la máquina (en euros).
- `cambio` : Cambio del que dispone la máquina. El cambio disponible se reduce cada vez que se devuelve al cliente la diferencia entre el crédito introducido y el precio del producto comprado. El cambio nunca se ve incrementado por las compras de los clientes.
- `recaudación` : Representa la suma de las ventas realizadas por la máquina (en euros). Se ve incrementada con cada nueva compra.
- Métodos:
- Constructor: `public Expendedora (double cambio, int stock, double precio)` . Crea la expendedora inicializando los atributos `cambio`, `stock` y `precio` con los valores indicados en los parámetros). El crédito y la recaudación serán cero.
- Consultores:
- Métodos consultores para los atributos crédito, cambio, y recaudación
- Los consultores para el `stock` y el `precio` los haremos previendo que en el futuro la máquina pueda expender más de un tipo de producto. Para consultar el `stock` y el `precio` se indicará como parámetro el número de producto que se quiere consultar aunque, por el momento se ignorará el valor de dicho atributo.
- `public getStock (int producto)` Devuelve el `stock` disponible del producto indicado. En esta versión simplificada se devolverá el valor del atributo `stock`, sea cual sea el valor de producto.
- `public getPrecio (int producto)` Devuelve el `precio` del producto indicado. En esta versión simplificada se devolverá el valor del atributo `precio`, sea cual sea el valor de producto.
- Modificadores: Para simplificar, consideramos que los atributos de la máquina solo van a cambiar por operaciones derivadas de su funcionamiento, por lo que no proporcionamos modificadores públicos
- Otros métodos:
- `public String toString()` Devuelve un `String` de la forma:

```

1  Credito: 3.00 euros
2  Cambio: 12.73 euros
3  Stock: 12 unidades
4  Recaudacion: 127.87 euros

```

- `public void introducirDinero(double importe)` Representa la operación mediante la cual el cliente añade dinero (crédito) a la máquina. Esta operación incrementa el crédito introducido por el cliente en el importe indicado como parámetro.
- `public double solicitarDevolucion()` Representa la operación mediante la cual el cliente solicita la devolución del crédito introducido sin realizar la compra. El método devuelve la cantidad de dinero que se devuelve al cliente.
- `public double comprarProducto(int producto) throws NoHayCambioException, NoHayProductoException, CreditoInsuficienteException`. Representa la operación mediante la cual el cliente selecciona un producto para su compra. El método devuelve la cantidad de dinero que se devuelve al cliente.

Si no se produce ninguna situación inesperada, se reduce el `stock` del producto, se devuelve el `cambio`, se pone el crédito a cero y se incrementa la `recaudación`.

Si la venta no es posible se lanzará la excepción correspondiente a la situación que impide completar la venta.

2. La clase `Producto` permite representar uno de los artículos de los que vende una máquina expendedora. Para ello utilizaremos tres atributos privados `nombre (String)`, `precio (double)` y `stock (int)`, y los siguientes métodos:
 - `public Producto(String nombre, double precio, int stock)` Constructor que inicializa el producto con los parámetros indicados
 - Consultores de los tres atributos: `getNombre`, `getPrecio` y `getStock`
 - `public int decrementarStock()`: Decrementa en 1 el `stock` del producto y devuelve el `stock` resultante.
3. La clase `TestExpendedora` sirve para probar los métodos desarrollados en las clases `Expendedora` y `Producto`.
 - Crea un Objeto de tipo `Expendedora` e inicializalo con: 12 unidades de stock, 5 euros de cambio y un precio de 3.75 euros. Muestra por pantalla su estado actual.
 - Simula la introducción por parte del cliente de un billete de 5 euros y muestra el estado de la máquina `Expendedora` .
 - Simula la compra de un `producto` y muestra la cantidad devuelta.
 - Simula la introducción de una moneda de 2 euros y solicita la devolución sin realizar ninguna compra y muestra la cantidad devuelta.
 - Intenta realizar una compra sin tener suficiente crédito y gestiona la excepción.
 - Crea otro objeto de tipo `Expendedora` que inicialmente tenga 0 unidades de stock (el resto de valores a tu gusto), simula la compra de un producto teniendo suficiente crédito y cambio. Gestiona la excepción.
 - Crea un último objeto de tipo `Expendedora` que inicialmente tenga 0 euros de cambio (el resto de valores a tu gusto), simula la compra de un producto para el que la máquina tenga que devolver algún importe, gestiona la excepción.

- Muestra las recaudaciones para las 3 máquinas expendedoras.
4. La clase `Surtido` representa una colección de productos. Para ello se usará un atributo `listaProductos`, array de `Productos`. El array se llenará con los datos de productos extraídos de un fichero de texto y, una vez creado el surtido no será posible añadir o quitar productos. Así, el array de productos estará siempre completo y no es necesario ningún atributo que indique cuantos productos hay en el array.

Se implementarán los siguientes métodos:

- `public Surtido() throws FileNotFoundException` Crea el surtido con los datos de los productos que se encuentran en el fichero `productos.txt`. El fichero tiene el siguiente formato:

```

1  <nº de productos>
2  <nombre de producto> <precio> <stock>
3  <nombre de producto> <precio> <stock>
4  <nombre de producto> <precio> <stock>
5  ...

```

Como vemos, la primera línea del fichero indica el número de productos que contiene el surtido. Este dato lo usaremos para dar al array de productos el tamaño adecuado. Ten en cuenta que la excepción `FileNotFoundException` hereda de `IOException`.

- `public int numProductos()` Devuelve el número de productos que componen el surtido
 - `public Producto getProducto(int numProducto)`: Devuelve el producto que ocupa la posición `numProducto` del surtido. La primera posición válida es la `1`. La posición `0` no se utiliza.
 - `public String[] getNombresProductos()` Devuelve un array con los nombres de los productos. La posición `0` del array no se utilizará (será `null`)
5. Crea una copia de la clase `Expededor` y llámala `ExpededorSurtido`. Añadir los atributos y hacer los cambios necesarios en la clase para que sea capaz de dispensar varios productos usando la nueva clase `Surtido`. Por ejemplo ya no tienen sentido los atributos `stock` y `precio` ya que pertenecen al `Surtido`. Modifica también el método `public String toString()`, para que muestre por pantalla el listado de productos con su nombre, precio y stock para mostrar al cliente que productos puede elegir. El código del producto coincidirá con su posición al leer el surtido.
6. Crea una copia de la clase `TestExpededor` y nómbralas `TestExpededor2` para adaptarla a los cambios hechos en la clase `Expededor` y usando la nueva posibilidad de comprar diferentes productos y usando solamente un único objeto `Expededor`. Al final en lugar de mostrar la recaudación de las 3 máquinas expendedoras, muestra solo la de la única que hay, que muestra el surtido de esta manera:

```

1  Prod.      Stock   Precio
2  CocaCola  18     1.5
3  Snickers  15     1.2
4  PotatoChips 25    1.0
5  Water      30     0.8
6  HamSandwich 10    2.5

```

22.2. Los flujos estándar

- (clase `LeeNombre`) Escribir un programa que solicite al usuario su nombre y, utilizando directamente `System.in`, lo lea de teclado y muestre por pantalla un mensaje del estilo "Su nombre es Miguel". Recuerda que `System.in` es un objeto de tipo `InputStream`. La clase `InputStream` permite **leer bytes** utilizando el método `read()`. Será tarea nuestra ir construyendo un `String` a partir de los bytes leídos. Prueba el programa de manera que el usuario incluya en su nombre algún carácter "extraño", por ejemplo el símbolo "`€`" ¿Funciona bien el programa? ¿Por qué?
- (clase `LeeEdad`) Escribir un programa que solicite al usuario su edad y, utilizando directamente `System.in`, lo lea de teclado y muestre por pantalla un mensaje del estilo "Su edad es 32 años". En este caso, será tarea nuestra construir un `String` a partir de los bytes leídos y transformarlo posteriormente en un entero.
- (clase `CambiarEstandar`). La salida estándar (`System.out`) y la salida de errores (`System.err`) están asociadas por defecto con la pantalla. Se puede cambiar este comportamiento por defecto utilizando los métodos `System.setOut` y `System.setErr` respectivamente. Investiga un poco cómo se utilizan, escribe un programa que asocie la salida estándar a al fichero `salida.txt` y la salida de errores al fichero `errores.txt` y, a continuación, escribe algún mensaje en cada uno de las salidas, por ejemplo `System.out.println("El resultado es 20");` y `System.err.println("ERROR: Elemento no encontrado");`
- (`SumarEdades`) - Escribir método `void sumaEdades()` que lea de teclado las edades de una serie de personas y muestre cuantos suman. El método finalizará cuando el usuario introduzca una edad negativa. - Escribir un método `main` que llame al método anterior para probarlo. - Modificar el método `main` de forma que, antes de llamar al método `sumaEdades`, se cambie la entrada estándar para que tome los datos del fichero `edades.txt` en lugar de leerlos de teclado.

22.3. InputStreamReader

1. (`leerByte`) `System.in` (`InputStream`) está orientado a lectura de bytes. Escribe un programa que lea un byte de teclado y muestre su valor (`int`) por pantalla. Pruébalo con un carácter "extraño", por ejemplo '€'.
2. (`leerCaracter`) `InputStreamReader` (`StreamReader`) está orientado a caracteres. Escribe un programa que lea un carácter de teclado usando un `InputStreamReader` y muestre su valor (`int`) por pantalla. Pruébalo con un carácter "extraño", por ejemplo '€'. ¿Se obtiene el mismo resultado que en el ejercicio anterior?

22.4. Entrada "orientada a líneas".

En los ejercicios anteriores, las limitaciones de la clase utilizada (`InputStream`), nos obliga a incluir en el programa instrucciones que detecten que el usuario ha terminado su entrada (ha pulsado **INTRO**). La clase `BufferedReader` dispone del método `readLine()`, capaz de leer una línea completa (la propia instrucción detecta el final de la línea) y devolver un `String`.

- 7.- Repite el ejercicio 1 utilizando un `BufferedReader` asociado a la entrada estándar. La clase `BufferedReader`, está orientada a leer caracteres en lugar de bytes. ¿Qué ocurre ahora si el usuario introduce un carácter "extraño" en su nombre?
- 8.- Repite el ejercicio 2 utilizando un `BufferedReader` asociado a la entrada estándar.

22.5. Lectura/escritura en ficheros

1. (EscribirFichero1) Escribe un programa que, usando las clases `FileOutputStream` y `FileInputStream`, (puedes usar más a parte de estas dos)
 - escriba tu nombre y altura en un fichero (`nombre.txt`).
 - lea el fichero creado y lo muestre por pantalla.
 - Si abrimos el fichero creado con un editor de textos, ¿su contenido es legible?
2. (EscribirFichero2) Repetir el ejercicio anterior utilizando las clases `FileReader` y `FileWriter` para generar el fichero `nombre2.txt`.

22.6. Uso de buffers

Los buffers hacen que las operaciones de lectura-escritura se realicen inicialmente en memoria y, cuando los buffers correspondientes están vacíos/llenos, se hagan definitivamente sobre el dispositivo.

1. (TestVelocidadBuffer) Vamos a probar la diferencia de tiempo que conlleva escribir datos a un fichero directamente o hacerlo a través de un buffer. Para ello, crea un fichero de 1 Mb (1000000 de bytes aprox.) usando un la clase `FileWriter` y mide el tiempo que tarda en crearlo. Posteriormente, crea un fichero de exactamente el mismo tamaño utilizando un `BufferedWriter` y mide el tiempo que tarda. ¿Hay diferencia?. Para medir el tiempo puedes utilizar `System.currentTimeMillis()`, inmediatamente antes y después de crear el fichero y restar los valores obtenidos.
2. (TestVelocidadBuffer2) Modifica el programa `TestVelocidadBuffer` para probar cómo afecta a la escritura con buffer la ejecución de la instrucción `flush()`. Esta instrucción fuerza el volcado del buffer a disco. ¿Disminuye la velocidad si tras cada operación de escritura ejecutamos `flush()`?
3. (TestVelocidadBuffer3) Modifica el programa `TestVelocidadBuffer` para probar cómo a la velocidad el tamaño del buffer. La clase `BufferedWriter` tiene un constructor que permite indicar el tamaño del buffer. Prueba con distintos valores.

22.7. Streams para información binaria

1. (Personas) Escribe un programa que, utilizando entre otras la clase `DataOutputStream`, almacene en un fichero llamado `personas.dat` la información relativa a una serie de personas que va introduciendo el usuario desde teclado:
 - `Nombre` (`String`)
 - `Edad` (`entero`)
 - `Peso` (`double`)
 - `Estatura` (`double`)
- La entrada del usuario terminará cuando se introduzca un nombre vacío.

Nota: Utiliza la clase `Scanner` para leer desde teclado y los métodos `writeDouble`, `writeInt` y `writeUTF` de la clase `DataOutput`/`InputStream` para escribir en el fichero)

Al finalizar el programa, abre el fichero resultante con un editor de texto (notepad o wordpad) ¿La información que contiene es legible?.

- (AñadirPersonas) Modifica el programa anterior para que el usuario, al comienzo del programa, pueda elegir si quiere añadir datos al fichero o sobre escribir la información que contiene.

Recomendación para Test

La pregunta debería ser del estilo de "¿Deseas añadir más datos? si eliges NO borraras el contenido anterior. ('S' para si o 'N' para no)"

El test usará los caracteres S o N para probar en ese sentido.

- (MostrarPersonas) Realizar un programa que lea la información del fichero `personas.dat` y la muestre por pantalla. Para determinar que no quedan más datos en el fichero podemos capturar la excepción `EOFException`

Recomendación para Test

La salida esperada debe ser similar a:

```
1 Registro 1: David 35 85,00 1,87
2 Registro 2: María 23 52,00 1,67
3 Registro 3: Pepe 46 67,00 1,80
4 Fichero leido completamente
```

- (CalculosPersonas) Realizar un programa, similar al anterior, que lea la información del fichero `personas.dat` y muestre por pantalla la estatura que tienen de media las personas cuya edad está entre 20 y 30 años.

Recomendación para Test

La salida esperada debe ser similar a:

```
1 Información sobre las personas de entre 20 y 30 años:
2 La media de estatura es: 1,67
```

22.8. Streams de objetos. Serialización.

1. Paquete `GuardaLeeLibros`

- (Autor) Crea la clase `Autor`, con los atributos **nombre**, **año de nacimiento** y **nacionalidad**. Incorpora un **constructor** que reciba todos los datos y el método `toString()`.
- (Libro) Crea la clase `Libro`, con los atributos **título**, **año de edición** y **autor** (Objeto de la clase `Autor`). Incorpora un **constructor** que reciba todos los datos y el método `toString()`.
- Escribe un programa `GuardaLibros` que cree tres libros y los almacene en el fichero `biblioteca.obj`.
- Las clases deberán implementar el interfaz `Serializable`.

2. Paquete `GuardaLeeLibros`

- Escribe un programa (`LeeLibros`) que lea los objetos del fichero `biblioteca.obj` y los muestre por pantalla (Libros y Autores).

22.9. Sockets

- Programar un Servidor que reciba una fecha (previamente validada por el cliente) y nos diga cual es nuestro signo del zodíaco occidental y el animal que corresponde en el zodíaco oriental (animales).

Test

Para poder pasar los tests correctamente la IP del servidor será localhost(127.0.0.1) y el PUERTO el 6000.

22.10. Más ejercicios (Lionel)

Muy Importante

Para probar algunos de estos ejercicios debes utilizar el archivo `Documentos.zip`. Descárgalo del aula virtual y descomprímelo en la carpeta de cada proyecto que crees.

Para superar los tests, además, se debe mantener la ruta de la carpeta Documentos en `files/_10_MasEjercicios/Documentos` dentro del proyecto de IntelliJ, como algunos ejercicios borran y moveran contenido, podrás descomprimir la carpeta `Documentos` a partir del archivo `Documentos.zip` que se incluye con el proyecto.

1. Mostrar información de ficheros

Implementa un programa que pida al usuario introducir por teclado una ruta del sistema de archivos (por ejemplo, `C:/Windows o Documentos`) y muestre información sobre dicha ruta (ver función más abajo). El proceso se repetirá una y otra vez hasta que el usuario introduzca una ruta vacía (tecla intro). Deberá manejar las posibles excepciones.

Necesitarás crear la función `void muestraInfoRuta(File ruta)` que dada una ruta de tipo `File` haga lo siguiente:

- Si es un archivo, mostrará por pantalla el nombre del archivo.
- Si es un directorio, mostrará por pantalla la lista de directorios y archivos que contiene (sus nombres). Deberá mostrar primero los directorios y luego los archivos.
- En cualquier caso, añade delante del nombre la etiqueta `[*]` o `[A]` para indicar si es un directorio o un archivo respectivamente.
- Si el path no existe lanzará un `FileNotFoundException`.

2. Mostrar información de ficheros (v2)

Partiendo de una copia del programa anterior, modifica la función `muestraInfoRuta`:

- En el caso de un directorio, mostrará la lista de directorios y archivos en orden alfabético. Es decir, primero los directorios en orden alfabético y luego los archivos en orden alfabético.
- Añade un segundo argumento `boolean info` que cuando sea `true` mostrará, junto a la información de cada directorio o archivo, su tamaño en bytes y la fecha de la última modificación. Cuando `info` sea `false` mostrará la información como en el ejercicio anterior.

3. Renombrando directorios y ficheros

Implementa un programa que haga lo siguiente:

- Cambiar el nombre de la carpeta `Documentos` a `DOCS`, el de la carpeta `Fotografias` a `FOTOS` y el de la carpeta `Libros` a `LECTURAS`
- Cambiar el nombre de todos los archivos de las carpetas `FOTOS` y `LECTURAS` quitándole la extensión. Por ejemplo, `astronauta.jpg` pasará a llamarse `astronauta`.

4. Creando (y moviendo) carpetas

Implementa un programa que cree, dentro de `Documentos`, dos nuevas carpetas: `MisCosas` y `Alfabeto`. Mueve las carpetas `Fotografias` y `Libros` dentro de `MisCosas`. Luego crea dentro de `Alfabeto` una carpeta por cada letra del alfabeto (en mayúsculas): `A`, `B`, `C`... `Z`. Te serán de ayuda los códigos numéricos ASCII: <https://elcodigoascii.com.ar>

5. Borrando archivos

Implementa un programa con una función `boolean borraTodo(File f)` que borre `f`: Si no existe lanzará una excepción. Si es un archivo lo borrará. Si es un directorio, borrará primero sus archivos y luego el propio directorio (recuerda que para poder borrar un directorio debe estar vacío). Devolverá `true` si pudo borrar el `File f` (`false` si no fué posible).

Prueba la función borrando las carpetas: `Documentos/Fotografias`, `Documentos/Libros` y `Documentos` (es decir, tres llamadas a la función, en ese orden).

Super extra challenge: Esta función, tal y como está definida, no borrará las subcarpetas que estén dentro de una carpeta (para ello habría que borrar primero el contenido de dichas subcarpetas). ¿Se te ocurre cómo podría hacerse?

6. Máximo y mínimo

Implementa un programa que muestre por pantalla los valores máximos y mínimos del archivo `numeros.txt`.

7. Notas de alumnos

El archivo `alumnos_notas.txt` contiene una lista de 10 alumnos y las notas que han obtenido en cada asignatura. El número de asignaturas de cada alumno es variable. Implementa un programa que muestre por pantalla la nota media de cada alumno junto a su nombre y apellido, ordenado por nota media de mayor a menor.

```

1 LISTADO DE NOTAS MEDIAS DE LOS ALUMNOS
2 -----
3 7,40 Toni Harper
4 6,50 Patrick Santos
5 6,25 Michele Poole
6 6,00 Troy Walters
7 5,83 Joanna Rogers
8 5,75 Ron Garza
9 5,43 Malcolm Lindsey
10 5,00 Gilbert Santiago
11 4,50 Gabriel Moreno
12 4,00 Vivian Chambers

```

8. Ordenando archivos

Implementa un programa que pida al usuario un nombre de archivo `A` para lectura y otro nombre de archivo `B` para escritura. Leerá el contenido del archivo `A` (por ejemplo `usa_personas.txt`) y lo escribirá ordenado alfabéticamente en `B` (por ejemplo `usa_personas_sorted.txt`).

9. Nombre y apellidos

Implementa un programa que genere aleatoriamente nombres de persona (combinando nombres y apellidos de `usa_nombres.txt` y `usa_apellidos.txt`). Se le pedirá al usuario cuántos nombres de persona desea generar y a qué archivo **añadirlos** (por ejemplo `usa_personas.txt`).

10. Diccionario

Implementa un programa que cree la carpeta `Diccionario` con tantos archivos como letras del abecedario (`A.txt`, `B.txt` ... `Z.txt`). Introducirá en cada archivo las palabras de `diccionario.txt` que comiencen por dicha letra.

11. Búsqueda en PI

Implementa un programa que pida al usuario un número de cualquier longitud, como por ejemplo "1234", y le diga al usuario si dicho número aparece en el primer millón de decimales del nº pi (están en el archivo `pi-million.txt`). No está permitido utilizar ninguna librería ni clase ni método que realice la búsqueda. Debes implementar el algoritmo de búsqueda tú.

12. Estadísticas

Implementa un programa que lea un documento de texto y muestre por pantalla algunos datos estadísticos: nº de líneas, nº de palabras, nº de caracteres y cuáles son las 10 palabras más comunes (y cuántas veces aparecen) `public static void muestraPalabrasMasComunes(Hashtable<String, Integer> t)`. Prueba el programa con los archivos de la carpeta `Libros`.

NOTA: Para llevar la cuenta de cuántas veces aparece cada palabra puedes utilizar una Colección que almacene pares de clave/valor. Por ejemplo `{“elefante”, 5}` o `{“casa”, 10}` son pares que asocian una palabra (clave) con un nº entero (valor).

```

1 ESTADÍSTICAS DE LIBROS
2 -----
3
4 Libro: quijote_cervantes.txt
5 Lineas totales: 37861
6 Número de palabras: 393764
7 Número de caracteres: 1723734
8 Las 10 palabras más comunes son:
9 que=19429
10 de=17988
11 y=15894
12 la=10200
13 a=9575
14 =9504
15 el=7957
16 en=7898
17 no=5611
18 se=4690
19
20 Libro: vida_unamuno.txt
21 Lineas totales: 10309
22 Número de palabras: 106585
23 Número de caracteres: 482163
24 Las 10 palabras más comunes son:
25 de=5355
26 la=4151
27 que=4056
28 y=3197
29 =2951
30 el=2679
31 en=2380
32 a=2096
33 es=1857
34 no=1671
35
36 Libro: lazarillo.txt
37 Lineas totales: 2504
38 Número de palabras: 23860
39 Número de caracteres: 102364
40 Las 10 palabras más comunes son:
41 y=1020
42 que=875
43 de=754
44 =695
45 a=525
46 la=524
47 el=449
48 en=410
49 no=325
50 con=270
51 ....

```

22.11. Aún más ejercicios

- (CuentaLineas) Escribe un programa que, sin utilizar la clase `Scanner` para contar las lineas, muestre el número de lineas que contiene un fichero de texto. El nombre del fichero se solicitará al usuario al comienzo de la ejecución (para esto si puedes usar la clase `Scanner`).
- (CuentaPalabras) Escribe un programa que, sin utilizar la clase `Scanner` para contar las lineas, muestre el número de palabras que contiene un fichero de texto. El nombre del fichero se solicitará al usuario al comienzo de la ejecución (para esto si puedes usar la clase `Scanner`).

Consejo

Lee el fichero, línea a línea y utiliza la clase `StringTokenizer` o bien el método `split` de la clase `String` para averiguar el nº de palabras.

- (censura) Escribir un programa que sustituya por otras, ciertas palabras de un fichero de texto. Para ello, se desarrollará y llamará al método `void aplicaCensura(String entrada, String censura, String salida)`, que lee de un fichero de entrada y mediante un fichero de censura, crea el correspondiente fichero modificado. Por ejemplo:

Fichero de entrada:

```
1 En un lugar de la Mancha, de cuyo nombre no quiero acordarme, no ha mucho tiempo que vivía un hidalgo de los de lanza en astillero
```

Fichero de censura:

```

1  lugar sitio
2  quiero debo
3  hidalgo noble

```

Fichero de salida:

```
1  En un sitio de la Mancha, de cuyo nombre no debo acordarme, no ha mucho tiempo que vivía un noble de los de lanza en astillero
```

Consejo

Valora la posibilidad de cargar el fichero de censura en un mapa o par clave, valor.

4. (`Concatenar1`) Escribe un programa que dados dos ficheros de texto `f1` y `f2` (introducidos por el usuario por teclado) confeccione un tercer fichero `f3` cuyo contenido sea el de `f1` y a continuación el de `f2`.
5. (`Concatenar2`) Escribe un programa que dados dos ficheros de texto `f1` y `f2` (introducidos por el usuario por teclado), añada al final de `f1` el contenido de `f2`. Es decir, como el ejercicio anterior, pero sin producir un nuevo fichero.
6. (`Iguales`) Escribir un programa que compruebe si el contenido de dos ficheros es idéntico. Puesto que no sabemos de qué tipo de ficheros se trata, (de texto, binarios, ...) habrá que hacer una comparación byte por byte.
7. Escribe los siguientes métodos y programa:
 - Método `void generar()` que genere 20 números aleatorios enteros entre 1 y 100 y los muestre por pantalla.
 - Método `void media()` que lea de teclado 20 números enteros y calcule su media.
 - Programa que, modificando la entrada y la salida estándar, llame a `generar()` para que los datos se graben en un fichero y a continuación llame a `media()` de manera que se tomen los datos del fichero generado.
8. (`Notas`) Escribir un programa que almacene en un fichero binario (`notas.dat`) las notas de 20 alumnos. El programa tendrá el siguiente funcionamiento:
 - En el fichero se guardarán como máximo 20 notas, pero se pueden guardar menos. El proceso de introducción de notas (y en consecuencia, el programa) finalizará cuando el usuario introduzca una nota no válida (menor que cero o mayor que 10).
 - Si, al comenzar la ejecución, el fichero ya contiene notas, se indicará al usuario cuántas faltan por añadir y las notas que introduzca el usuario se añadirán a continuación de las que hay.
 - Si, al comenzar la ejecución, el fichero ya contiene 20 notas, se le preguntará al usuario si desea sobrescribirlas. En caso afirmativo las notas que introduzca sustituirán a las que hay y en caso negativo el fichero no se modificará.

22 de febrero de 2026

23. 7.4 Talleres

23.1. Taller UD06_01: GitHub Classroom

23.1.1. Unirnos a GitHub Classroom

Aceptamos el *Assignement* (la tarea/ejercicio) a partir del link del profesor, en este caso: <https://classroom.github.com/a/sbfeTX2U>

23.1.2. Tarea

Debes enviar tus soluciones a GitHub Classroom y superar al menos la mitad de los tests, cuantos más tests superados, mejor nota tendrás en la tarea.

 25 de enero de 2026

23.2. Taller UD06_T02: Sockets en la nube (AWS)

La intención de este documento es la de dar una perspectiva más realista del uso de sockets, ya que en lugar de usar la misma máquina del alumno como cliente y servidor, vamos a desplegar el servidor del socket en una máquina alojada en la nube de Amazon (AWS).

23.2.1. Requisitos

Para realizar esta práctica guiada necesitamos:

- Acceso al Learner Lab proporcionado por el profesor. (<https://awsacademy.instructure.com>)
- Conocimientos sobre los sockets, IP's y puertos.
- Un dispositivo local con capacidad de ejecutar un cliente de socket, con acceso a los puertos e Ip's de AWS (Ojo con la red de conselleria)

23.2.2. Guía paso a paso

23.2.2.1. PREPARAR EL ENTORNO DE LA NUBE

23.2.2.1.1. Iniciar Laboratorio

Lo primero que necesitamos es arrancar el laboratorio, para ello Accedemos al LMS del awsacademy, buscamos el Curso facilitado por el docente, accedemos a sus contenidos y a continuación al Learner Lab. (Si es la primera vez que accedemos debemos aceptar los términos de uso).

Inicialmente el laboratorio está en rojo:



Elegimos la opción `Start Lab` y esperamos a que aparezca el laboratorio en verde:



Por defecto el Learner Lab nos proporciona 100 50 dolares de saldo, y un tiempo de 4 horas, tras el cual se detendrán la mayoría de servicios que tengamos en marcha. Pero mientras quede saldo podemos volver a iniciar el Laboratorio y dispondremos de 4 horas más.

Una vez aparece en verde podemos hacer click sobre las letras AWS y aparecerá el Dashboard de AWS (debemos permitir las ventanas emergentes):

The screenshot shows the AWS Home Dashboard. At the top, there's a search bar with 'Buscar' and a placeholder '[Alt+S]'. To the right are icons for notifications, help, and account information ('Norte de Virginia'). The URL in the address bar is 'voclabs/user2456362=Estudiante_de_prueba @ 4962-3004-7969'.

Recent Visits: A section titled 'Visitados recientemente' with a link to 'Información'. It shows a large icon of a cube.

Unvisited Services: A section titled 'Servicios no visitados recientemente' with a link to 'Información'. It says 'Explore uno de estos servicios de AWS visitados habitualmente.' Below are links for IAM, EC2, S3, RDS, and Lambda.

Welcome Section: 'Le damos la bienvenida a AWS' with links to 'Introducción a AWS' (with a rocket icon) and 'Formación y certificación' (with a chart icon).

AWS Health: Shows 'Problemas abiertos' (0) and 'Cambios programados' (0). It includes links for 'Últimos 7 días' and 'Próximos 7 días y últimos 7 días'.

Footer: 'Ver todos los servicios'

23.2.2.1.2. Crear un entorno Cloud9

Cloud9 es un entorno de desarrollo en la nube que proporciona AWS asociado a una instancia EC2 (máquina virtual en la nube). El primer paso es crear este entorno, para ello buscamos cloud9 en la parte superior del Dashboard:

The screenshot shows the AWS search results for 'cloud9'. The search bar at the top has 'cloud9' entered. The results are categorized under 'Servicios' (5), 'Características' (5), 'Recursos' (New), 'Blogs' (39), 'Documentación' (37.748), and 'Artículos de conocimiento'. The 'Cloud9' service result is highlighted with a red oval. It has a star icon and the text 'Un IDE en la nube para escribir, ejecutar y depurar código'. There's also a link 'Ver los 5 resultados ▶'.

A continuación seleccionamos Create environment :

En la siguiente ventana debemos especificar el **nombre** (Name), cambiaremos la **plataforma** a Ubuntu Server 18.04 LTS , también podemos ampliar el tiempo de Timeout para no tener problemas a 4 horas i por último dentro de los **Network settings** elegiremos la conexión por `SSH` , el resto de opciones se quedan por defecto y pulsamos el botón naranja del final `Create` .

Crear entorno Info

Detalles

Nombre
cloud9David
Límite de 60 caracteres alfanuméricos y únicos por usuario.

Descripción: *opcional*
Límite de 200 caracteres.

Tipo de entorno Info
Determina en qué se ejecutará el IDE de Cloud9.

- Nueva instancia de EC2**
Cloud9 crea una instancia de EC2 en su cuenta. Cloud9 no puede cambiar la configuración de la instancia de EC2 después de crearla.
- Computación existente**
Ya tiene una instancia o un servidor que desea usar.

Nueva instancia de EC2

Tipo de Instancia Info
La memoria y la CPU de la instancia de EC2 que se creará para que se ejecute Cloud9.

- t2.micro (1 GiB RAM + 1 vCPU)**
Apto para el nivel gratuito. Ideal para usuarios educativos y de exploración.
- t3.small (2 GiB RAM + 2 vCPU)**
Recomendado para proyectos web pequeños.
- m5.large (8 GiB RAM + 2 vCPU)**
Recomendado para la producción y el desarrollo de uso más general.

Tipos de instancias adicionales
Explore instancias adicionales que se ajusten a sus necesidades.

Plataforma [Info](#)
 Se instalará en su instancia de EC2. Recomendamos Amazon Linux 2023.

Ubuntu Server 22.04 LTS

Tiempo de espera
 Cuánto tiempo puede permanecer inactivo Cloud9 (sin intervención del usuario) antes de hibernar automáticamente. Esto ayuda a evitar cargos innecesarios.

4 horas

Configuración de red [Info](#)

Conexión
 Cómo se accede a su entorno.

AWS Systems Manager (SSM)
 Accede al entorno a través de SSM sin abrir los puertos entrantes (sin entrada).

Secure Shell (SSH)
 Accede al entorno directamente a través de SSH, abre los puertos entrantes.

▶ **Configuración de VPC** [Info](#)

▶ **Etiquetas: opcional** [Info](#)
 Las etiquetas se asignan a los recursos de AWS. Cada etiqueta consta de una clave y un valor opcional. Puede utilizar etiquetas para buscar y filtrar sus recursos o realizar un seguimiento de sus costos de AWS.

i **Se crearán los siguientes recursos de IAM en su cuenta**

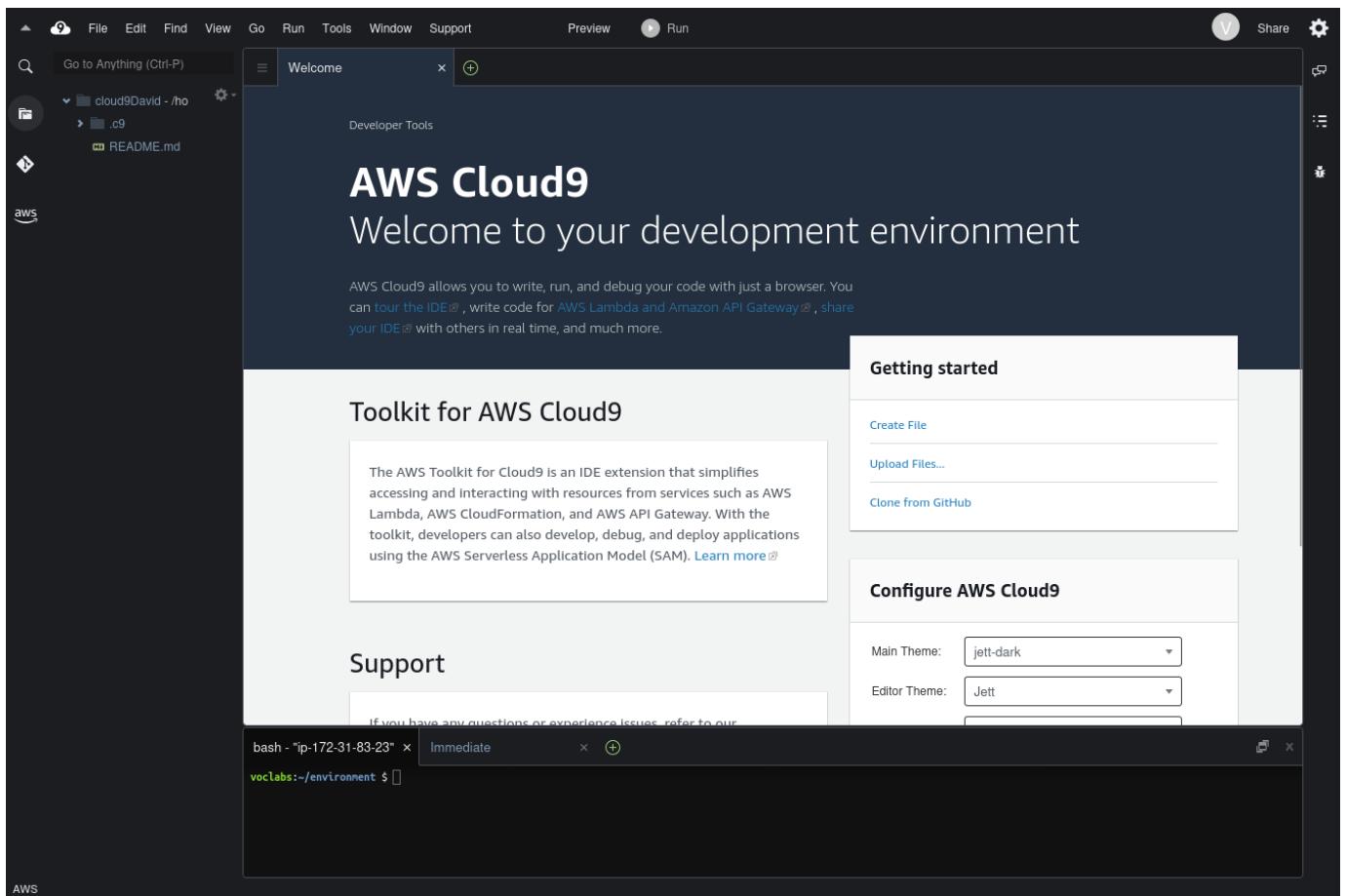
- **AWSServiceRoleForAWSCloud9**- AWS Cloud9 crea un rol vinculado a un servicio para usted. Esto permite que AWS Cloud9 invoque a otros servicios de AWS en su nombre. Puede eliminar el rol desde la consola de AWS IAM cuando ya no tenga ningún entorno de AWS Cloud9. [Más información](#)

Cancelar **Crear**

Si todo ha ido bien podemos seleccionar el botón `Open`:

Environments (1)						
Delete View details Open in Cloud9 Create environment						
Name	Cloud9 IDE	Environment type	Connection	Permission	Owner ARN	
cloud9David	Open	EC2 Instance	Secure Shell (SSH)	Owner	arn:aws:sts::496230047969:assumed-role/voclabs/user2456362=Estudiante_de_prueba	

Y deberíamos ver algo parecido a esto:



23.2.2.1.3. Creación del servidor de Sockets

Primero cerraremos la ventana de bienvenida, a continuación creamos un nuevo fichero, por ejemplo `ServidorSocket.java` con el siguiente código java:

```

1 import java.io.*;
2 import java.net.*;
3
4 public class ServidorSocket {
5
6     private static final int PORT=11000;
7
8     public static void main(String[] args) throws IOException, ClassNotFoundException {
9         String FraseClient;
10        String FraseMajuscules;
11        ServerSocket serverSocket;
12        Socket clientSocket;
13        ObjectInputStream entrada;
14        ObjectOutputStream eixida;
15        serverSocket = new ServerSocket(PORT);
16        System.out.println("Server iniciado y escuchando en el puerto "+ PORT);
17        while (true) {
18            clientSocket = serverSocket.accept();
19            entrada = new ObjectInputStream(clientSocket.getInputStream());
20            FraseClient = (String) entrada.readObject();
21
22            System.out.println("La frase recibida es: " + FraseClient);
23
24            eixida = new ObjectOutputStream(clientSocket.getOutputStream());
25            FraseMajuscules = FraseClient.toUpperCase();
26            System.out.println("El server devuelve la frase: " + FraseMajuscules);
27            eixida.writeObject(FraseMajuscules);
28
29            clientSocket.close();
30            System.out.println("Server esperando una nueva conexión...");
31        }
32    }
33 }
```

Debería quedar algo así:

```

1 import java.io.*;
2 import java.net.*;
3
4 public class ServidorSocket {
5
6     private static final int PORT=11000;
7
8     public static void main(String[] args) throws IOException, ClassNotFoundException {
9         String FraseClient;
10        String FraseMajuscules;
11        ServerSocket serverSocket;
12        Socket clientSocket;
13        ObjectInputStream entrada;
14        ObjectOutputStream eixida;
15        serverSocket = new ServerSocket(PORT);
16        System.out.println("Server iniciado y escuchando en el puerto " + PORT);
17        while (true) {
18            clientSocket = serverSocket.accept();
19            entrada = new ObjectInputStream(clientSocket.getInputStream());
20            FraseClient = (String) entrada.readObject();
21            System.out.println("La frase recibida es: " + FraseClient);
22
23            eixida = new ObjectOutputStream(clientSocket.getOutputStream());
24            FraseMajuscules = FraseClient.toUpperCase();
25            System.out.println("El server devuelve la frase: " + FraseMajuscules);
26            eixida.writeObject(FraseMajuscules);
27
28            clientSocket.close();
29            System.out.println("Server esperando una nueva conexión...");
30        }
31    }
32 }
33

```

bash - "ip-172-31-83-23" x Immediate x +
voclabs:~/environment \$

Y si iniciamos el servidor:

```

bash - "ip-172-31-83-23" x Immediate x ServidorSocket.java - Run
Stop Command: ServidorSocket.java
Building ServidorSocket.java and running ServidorSocket
Server iniciado y escuchando en el puerto 11000

```

23.2.2.1.4. Abrir el puerto en la instancia EC2 del cloud9

Ahora debemos volver a la pestaña donde tenemos el Dashboard de AWS y buscar EC2 (donde antes buscamos cloud9):

AWS Cloud9

Resultados de la búsqueda de "ec2"

Servicios (11)

Características (51)

Recursos New

Blogs (36)

Documentación (102.796)

Artículos de conocimiento (30)

Tutoriales (19)

Servicios

Ver los 11 resultados ▾

EC2 ☆
Servidores virtuales en la nube

EC2 Image Builder ☆
Un servicio administrado para automatizar la creación, personalización e implementaci...

AWS Outposts ☆

Una vez abierto elegimos la opción **Instancias (en ejecución)**:

The screenshot shows the AWS EC2 Resources page. On the left, there's a sidebar with options like 'Panel de EC2', 'Instancias', and 'Auto Scaling Groups'. The main area displays a summary of resources:

- Instancias (en ejecución):** 1
- Balanceadores de carga:** 0
- Grupos de seguridad:** 2
- Auto Scaling Groups:**
- Direcciones IP elásticas:**
- Grupos de ubicación:**

Deberíamos tener al menos una Instancia, si tenemos más de una debemos buscar la que contenga el nombre de nuestra instancia cloud9, debemos marcar el check que tiene justo delante del nombre y a continuación elegir la pestaña Seguridad :

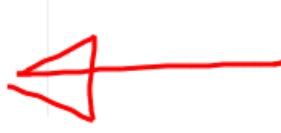
The screenshot shows the AWS EC2 Instances details page for a single instance. The instance is named "aws-cloud9-cloud9David-83bd4fb6e5bf401e9...". The "Seguridad" tab is selected. The instance details include:

- Estado de la instancia: En ejecución
- ID de la instancia: i-069949e33eb20baa8
- Tipo de instancia: t2.micro
- Comprobación: 2/2 comprobaciones

Si nos fijamos en las reglas de entrada del grupo de seguridad, solo tiene habilitada la entrada para el puerto 22 (SSH), a continuación hacemos click sobre el enlace del Grupo de seguridad:

Grupos de seguridad

[sg-0ab93f95e05631a5c \(aws-cloud9-cloud9David-83bd4fb6e5bf401e9b81d1e6f35291c5-InstanceSecurityGroup-7JT461ZI2RR\)](#)



▼ Reglas de entrada

Filtrar reglas			
Nombre	ID de la regla del grupo d...	Intervalo de pu...	Protocolo
-	sgr-0cb10956d0b223f9d	22	TCP
-	sgr-09bd2851a6631f7da	22	TCP

Y añadiremos el puerto 11000 (o el que hayamos elegido para nuestro servidor) a las reglas de entrada, elegimos el botón **Editar reglas de entrada**, a continuación **Agregar regla**. Elegimos **TCP Personalizado**, puerto 11000 y **AnywhereIpv4** y añadimos una descripción si lo deseamos:

Editar reglas de entrada Información

Las reglas de entrada controlan el tráfico entrante que puede llegar a la instancia.

ID de la regla del grupo de seguridad	Tipo	Información	Protocolo	Intervalo de puertos	Origen	Descripción: opcional
sgr-0cb10956d0b223f9d	SSH		TCP	22	Personalizada	<input type="text"/> 35.172.155.96/27
sgr-09bd2851a6631f7da	SSH		TCP	22	Personalizada	<input type="text"/> 35.172.155.192/27
-	TCP personalizado		TCP	11000	Anywhere-Ipv4	<input type="text"/> SocketServer
						<input type="text"/> 0.0.0.0/0

Agregar regla

Cancelar **Previsualizar los cambios** **Guardar reglas**

Una vez hecho esto si volvemos a la pestaña Seguridad de nuestra instancia EC2 veremos la regla añadida.

23.2.2.1.5. Dirección pública de la EC2

Necesitamos saber la DNS de IPv4 pública de nuestra instancia EC2 para acceder desde el cliente, marcamos el check de nuestra instancia y accedemos a la primera pestaña **Detalles**, y nos fijamos en la parte derecha y pulsaremos el botón de copiar y guardaremos esta información para más adelante:

Instancias (1/1) Información

Name | ID de la instancia | Estado de la i... | Tipo de inst... | Comprobación ... | Estado de la ... | Zona de dispon... | DNS de IPv4 pública | Dirección IP...

aws-cloud9-cloud9David-83bd4fb6e5bf401e9b81d1e6f35291c5 | i-069949e33eb20baa8 | En ejecución | t2.micro | 2/2 comprobador | Sin alarmas | us-east-1c | ec2-3-84-52-97.compute...

Lanzar instancias

Instancia: i-069949e33eb20baa8 (aws-cloud9-cloud9David-83bd4fb6e5bf401e9b81d1e6f35291c5)

Detalles | Seguridad | Redes | Almacenamiento | Comprobaciones de estado | Monitoreo | Etiquetas

Resumen de instancia Información

ID de la instancia <input checked="" type="checkbox"/> i-069949e33eb20baa8 (aws-cloud9-cloud9David-83bd4fb6e5bf401e9b81d1e6f35291c5)	Dirección IPv4 pública <input type="checkbox"/> 3.84.52.97 dirección abierta	Direcciones IPv4 privadas <input type="checkbox"/> 172.31.83.23
Dirección IPv6 -	Estado de la instancia <input checked="" type="checkbox"/> En ejecución	DNS de IPv4 pública <input type="checkbox"/> ec2-3-84-52-97.compute-1.amazonaws.com dirección abierta
Tipo de nombre de anfitrión Nombre de IP: ip-172-31-83-23.ec2.internal	Nombre DNS de IP privada (solo IPv4) <input type="checkbox"/> ip-172-31-83-23.ec2.internal	Direcciones IP elásticas -
Responder al nombre DNS de recurso privado -	Tipo de instancia t2.micro	Hallazgo de AWS Compute Optimizer <input type="checkbox"/> Suscribirse a AWS Compute Optimizer para recibir recomendaciones.
Dirección IP asignada automáticamente <input type="checkbox"/> 3.84.52.97 [IP pública]	ID de VPC <input type="checkbox"/> vpc-0ba927e3f4ec4d112	

23.2.2.2. TAREA 2: PREPARAR EL CLIENTE LOCAL

En nuestro IDE preferido creamos un nuevo archivo `ClienteSocket.java` con el siguiente código:

```

1 import java.io.*;
2 import java.net.*;
3 import java.util.Scanner;
4
5 public class ClienteSocket {
6
7     private static final String DNSAWS = "ec2-3-84-52-97.compute-1.amazonaws.com";
8
9     public static void main(String[] args) throws IOException, ClassNotFoundException {
10         Socket socket;
11         ObjectInputStream entrada;
12         ObjectOutputStream eixida;
13         String frase;
14
15         socket = new Socket(DNSAWS, 11000);
16         eixida = new ObjectOutputStream(socket.getOutputStream());
17
18         System.out.println("Introduce la frase a enviar en minúsculas");
19         Scanner in = new Scanner(System.in);
20         frase = in.nextLine();
21         System.out.println("Se envia la frase " + frase);
22         eixida.writeObject(frase);
23
24         entrada = new ObjectInputStream(socket.getInputStream());
25         System.out.println(
26             "La frase recibida es: " + (String) entrada.readObject());
27         socket.close();
28     }
29 }
```



Recuerda cambiar la constante `DNSAWS` por el `String` que corresponde con la dirección DNS IPv4 de tu instancia EC2 obtenida en el punto 3.1.5.

23.2.2.3. EJECUCIÓN DE PRUEBA

23.2.2.3.1. Desde el punto de vista del cliente

Una vez ejecutado el cliente debe aparecer algo similar a esto:

```
1 Introduce la frase a enviar en minúsculas
```

Escribimos nuestra frase, y al pulsar INTRO obtenemos el siguiente resultado:

```

1 Introduce la frase a enviar en minúsculas
2 esta frase está en minúsculas
3 Se envia la frase esta frase está en minúsculas
4 La frase recibida es: ESTA FRASE ESTÁ EN MINÚSCULAS
```

23.2.2.3.2. Desde el punto de vista del servidor

La consola de salida del servidor por su parte debe haber registrado la conexión del cliente, la recepción de la frase, y la frase devuelta:

```

1 Server iniciado y escuchando en el puerto 11000
2 La frase recibida es: esta frase está en minúsculas
3 El server devuelve la frase: ESTA FRASE ESTÁ EN MINÚSCULAS
4 Server esperando una nueva conexión...
```

23.2.3. Tarea

Genera un pequeño documento con explicaciones donde muestres una captura de pantalla donde se vea el servidor después de haber esperado la conexión, haber recibido los datos del cliente y devuelto la información procesada. Adjunta también explicaciones y la captura del mismo ejemplo desde el punto de vista del cliente. El ejemplo debe incluir tu nombre y apellidos en la información enviada y recibida.

Convierte el documento en pdf y adjúntalo a la tarea de Aules de la UD06.

Q2 de febrero de 2026

8. UD07

24. 8.1 Colecciones



24.1. Introducción

Cuando el volumen de datos a manejar por una aplicación es elevado, no basta con utilizar variables. Manejar los datos de un único pedido en una aplicación puede ser relativamente sencillo, pues un pedido está compuesto por una serie de datos y eso simplemente se traduce en varias variables.

Pero, ¿qué ocurre cuando en una aplicación tenemos que gestionar varios pedidos a la vez? Lo mismo ocurre en otros casos. Para poder realizar ciertas aplicaciones se necesita poder manejar datos que van más allá de meros datos simples (números y letras). A veces, los datos que tiene que manejar la aplicación son datos compuestos, es decir, datos que están compuestos a su vez de varios datos más simples. Por ejemplo, un pedido está compuesto por varios datos, los datos podrían ser el cliente que hace el pedido, la dirección de entrega, la fecha requerida de entrega y los artículos del pedido.

Los datos compuestos son un tipo de estructura de datos, y en realidad ya los has manejado. Las clases son un ejemplo de estructuras de datos que permiten almacenar datos compuestos, y el objeto en sí, la instancia de una clase, sería el dato compuesto. Pero, a veces, los datos tienen estructuras aún más complejas, y son necesarias soluciones adicionales.

Aquí podrás aprender esas soluciones adicionales. Esas soluciones consisten básicamente en la capacidad de poder manejar varios datos del mismo o diferente tipo de forma dinámica y flexible.

24.2. Estructuras de almacenamiento

¿Cómo almacenarías en memoria un listado de números del que tienes que extraer el valor máximo?

Seguro que te resultaría fácil. Pero, ¿y si el listado de números no tiene un tamaño fijo, sino que puede variar en tamaño de forma dinámica? Entonces la cosa se complica.

Un listado de números que aumenta o decrece en tamaño es una de las cosas que aprenderás a utilizar aquí, utilizando estructuras de datos.

Pasaremos por alto las clases y los objetos, pues ya los has visto con anterioridad, pero debes saber que las clases en sí mismas son la evolución de un tipo de estructuras de datos conocidas como datos compuestos (también llamadas registros). Las clases, además de aportar la ventaja de agrupar datos relacionados entre sí en una misma estructura (característica aportada por los datos compuestos), permiten agregar métodos que manejen dichos datos, ofreciendo una herramienta de programación sin igual. Pero todo esto ya lo sabías.

Las estructuras de almacenamiento, en general, se pueden clasificar de varias formas. Por ejemplo, atendiendo a **si pueden almacenar datos de diferente tipo, o si solo pueden almacenar datos de un solo tipo**, se pueden distinguir:

- **Estructuras con capacidad de almacenar varios datos del mismo tipo:** varios números, varios caracteres, etc. Ejemplos de estas estructuras son los arrays, las cadenas de caracteres, las listas y los conjuntos.
- **Estructuras con capacidad de almacenar varios datos de distinto tipo:** números, fechas, cadenas de caracteres, etc., todo junto dentro de una misma estructura. Ejemplos de este tipo de estructuras son las clases.

Otra forma de clasificar las estructuras de almacenamiento va **en función de si pueden o no cambiar de tamaño** de forma dinámica:

- **Estructuras cuyo tamaño se establece en el momento de la creación o definición y su tamaño no puede variar después.** Ejemplos de estas estructuras son los arrays y las matrices (arrays multimensionales).
- **Estructuras cuyo tamaño es variable (conocidas como estructuras dinámicas). Su tamaño crece o decrece según las necesidades de forma dinámica.** Es el caso de las listas, árboles, conjuntos y, como veremos también, el caso de algunos tipos de cadenas de caracteres.

Por último, atendiendo a **la forma en la que los datos se ordenan** dentro de la estructura, podemos diferenciar varios tipos de estructuras:

- **Estructuras que no se ordenan de por sí**, y debe ser el programador el encargado de ordenar los datos si fuera necesario. Un ejemplo de estas estructuras son los arrays.
- **Estructuras ordenadas.** Se trata de estructuras que al incorporar un dato nuevo a todos los datos existentes, este se almacena en una posición concreta que irá en función del orden. El orden establecido en la estructura puede variar dependiendo de las necesidades del programa: alfabético, orden numérico de mayor a menor, momento de inserción, etc.

Todavía no conoces mucho de las estructuras, y probablemente todo te suena raro y extraño. No te preocupes, poco a poco irás descubriendolas. Verás que son sencillas de utilizar y muy cómodas.

24.3. Clases y métodos genéricos

¿Crees qué el código es más legible al utilizar genéricos o qué se complica? La verdad es que al principio cuesta, pero después, el código se entiende mejor que si se empieza a insertar conversiones de tipo.

Las clases genéricas son equivalentes a los métodos genéricos pero a nivel de clase, permiten definir un parámetro de tipo o genérico que se podrá usar a lo largo de toda la clase, facilitando así crear clases genéricas que son capaces de trabajar con diferentes tipos de datos base. Para crear una clase genérica se especifican los parámetros de tipo al lado del nombre e de la clase:

```

1  public class Util<T> {
2      T t1;
3      public void invertir(T[] array) {
4          for (int i = 0; i < array.length / 2; i++) {
5              t1 = array[i];
6              array[i] = array[array.length - i - 1];
7              array[array.length - i - 1] = t1;
8          }
9      }
10 }
```

En el ejemplo anterior, la clase `Util` contiene el método `invertir` cuya función es invertir el orden de los elementos de cualquier `array`, sea del tipo que sea. Para usar esa clase genérica hay que crear un objeto o instancia de dicha clase especificando el tipo base entre los símbolos menor que ("<") y mayor que (">"), justo detrás del nombre e de la clase. Veamos un ejemplo:

```

1  Integer[] numeros={0,1,2,3,4,5,6,7,8,9};
2  Util<Integer> u= new Util<Integer>();
3  u.invertir(numeros);
4  for (int i=0;i<numeros.length;i++){
5      System.out.println(numeros[i]);
6  }
```

Como puedes observar, el uso de genéricos es sencillo, tanto a nivel de clase como a nivel de método.

Simplemente, a la hora de crear una instancia de una clase genérica, hay que especificar el tipo, tanto en la definición (`Util<Integer> u`) como en la creación (`new Util<Integer>()`).

Los genéricos los vamos a usar ampliamente a partir de ahora, aplicados a un montón de clases genéricas que tiene Java y que son de gran utilidad, por lo que es conveniente que aprendas bien a usar una clase genérica.

Acción

Los parámetros de tipo de las clases genéricas solo pueden ser clases, no pueden ser jamás tipos de datos primitivos como `int`, `short`, `double`, etc. En su lugar, debemos usar sus clases envoltorio (wrappers) `Integer`, `Short`, `Double`, etc.

Todavía hay un montón de cosas más sobre los métodos y las clases genéricas que deberías saber. A continuación se muestran algunos usos interesantes de los genéricos:

- Dos o más parámetros de tipo (I):

```
1 public class Util<T,M>{
2     public static <T,M> int sumaDeLongitudes (T[] a, M[] b){
3         return a.length+b.length;
4     }
5 }
```

Información

Si el método genérico necesita tener dos o más parámetros genéricos, podemos indicarlo separándolos por comas. En el ejemplo anterior se suman las longitudes de dos arrays que no tienen que ser del mismo tipo.

- Dos o más parámetros de tipo (II):

```
1 Integer[] a1={0,1,2,3,4};
2 Double[] a2={0d,1d,2d,3d,4d};
3 int resultado=Util.<Integer,Double>sumaDeLongitudes(a1,a2);
4 System.out.println(resultado);
```

Información

Usar un método o una clase con dos o más parámetros genéricos es sencillo, a la hora de invocar al método o crear la clase, se indican los tipos base separados por coma.

- Dos o más parámetros de tipo (III):

```
1 public class Terna <A,B,C>{
2     A a;
3     B b;
4     C c;
5     public terna(A a, B b, C c){
6         this.a=a;
7         this.b=b;
8         this.c=c;
9     }
10    public A getA(){return a;}
11    public B getB(){return b;}
12    public C getC(){return c;}
13 }
```

Información

Si una clase genérica necesita tener dos o más parámetros genéricos, podemos indicarlo separándolos por comas. En el ejemplo anterior se muestra una clase que almacena una terna de elementos de diferente tipo base que están relacionados entre sí.

- Métodos con tipos adicionales:

```

1 class Util<A,B>{
2     A a;
3     Util (A a){
4         this.a=a;
5     }
6     public <B> void Salida(B b){
7         System.out.println(a.toString() + b.toString());
8     }
9 }
```

Información

Una clase genérica puede tener unos parámetros genéricos, pero si en uno de sus métodos necesitamos otros parámetros genéricos distintos, no hay problema, podemos combinarlos.

- Inferencia de tipos (I):

```

1 Integer[] a1={0,1,2,3,4};
2 Double[] a2={0d,1d,2d,3d,4d};
3 util.<Integer,Double>sumadeLongitudes(a1,a2);
4 util.sumaDeLongitudes(a1,a2);
```

Información

No siempre es necesario indicar los tipos a la hora de instanciar un método genérico. A partir de Java 7, es capaz de determinar los tipos a partir de los parámetros. Las dos expresiones de arriba serían válidas y funcionarían. Si no es capaz de inferirlos, nos dará un error a la hora de compilar.

- Inferencia de tipos (II):

```

1 Integer a1=0;
2 Double d1=1.3d;
3 Float f1=1.4f;
4 Terna <Integer,Double,Float> t=new Terna<>(a1,d1,f1);
```

Información

A partir de Java 7 es posible usar el operador diamante `<>` para simplificar la instancia o creación de nuevos objetos a partir de clases genéricas. **Cuidado, esto solo es posible a partir de Java 7.**

- Limitación de tipos

```

1 public class Util {
2     public static <T extends Number> Double sumar (T t1, T t2){
3         return new Double(t1.doubleValue() + t2.doubleValue());
4     }
5 }
```

Información

Se pueden limitar el conjunto de tipos que se pueden usar con una clase o método genérico usando el operador `extends`. El operador `extends` permite indicar que la clase que se pasa como parámetro genérico tiene que derivar de una clase específica. En el ejemplo, no se admitirá ninguna clase que no derive de `Number`, pudiendo así realizar operaciones matemáticas.

- Paso de clases genéricas por parámetro

```

1 public class Ejemplo <A> {
2     public A a;
3 }
4 ...
5 void test (Ejemplo<Integer> e) {
6     ...
7 }
```

Información

Cuando un método tiene como parámetro una clase genérica (como en el caso del método test del ejemplo), se puede especificar cual debe ser el tipo base usado en la instancia de la clase genérica que se le pasa como argumento. Esto permite, entre otras cosas, crear diferentes versiones de un mismo método (sobrecarga), dependiendo del tipo base usado en la instancia de la clase genérica se ejecutarán una versión u otra.

- Paso de clases genéricas por parámetro. Wildcards. (I)

```

1  public class Ejemplo <A> {
2      public A a;
3  }
4  ...
5  void test (Ejemplo<?> e) {
6      ...
7 }
```

Información

Cuando un método admite como parámetro una clase genérica en la que no importa el tipo de objeto sobre la que se ha creado, podemos usar el interrogante para indicar "*cualquier tipo*".

- Paso de clases genéricas por parámetro. Wildcards. (II)

```

1  public class Ejemplo <A> {
2      public A a;
3  }
4  ...
5  void test (Ejemplo<? extends Number> e) {
6      ...
7 }
```

Información

También es posible limitar el conjunto de tipos que una clase genérica puede usar, a través del operador `extends`. El ejemplo anterior es como decir "*cualquier tipo que derive de Number*"

24.4. Colecciones

24.4.1. Introducción

¿Qué consideras una colección? Pues seguramente al pensar en el término se te viene a la cabeza una colección de libros o algo parecido, y la idea no va muy desencaminada. **Una colección a nivel de software es un grupo de elementos almacenados de forma conjunta en una misma estructura.** Eso son las colecciones.

Las colecciones definen un conjunto de interfaces, clases genéricas y algoritmos que permiten manejar grupos de objetos, todo ello enfocado a potenciar la reusabilidad del software y facilitar las tareas de programación. Te parecerá increíble el tiempo que se ahorra empleando colecciones y cómo se reduce la complejidad del software usándolas adecuadamente. Las colecciones permiten almacenar y manipular grupos de objetos que, a priori, están relacionados entre sí (aunque no es obligatorio que estén relacionados, lo lógico es que si se almacenan juntos es porque tienen alguna relación entre sí), pudiendo trabajar con cualquier tipo de objeto (de ahí que se empleen los genéricos en las colecciones).

Además las colecciones permiten realizar algunas operaciones útiles sobre los elementos almacenados, tales como búsqueda u ordenación. En algunos casos es necesario que los objetos almacenados cumplan algunas condiciones (que implementen algunas interfaces), para poder hacer uso de estos algoritmos.

Las colecciones son en general elementos de programación que están disponibles en muchos lenguajes de programación. En algunos lenguajes de programación su uso es algo más complejo (como es el caso de C++), pero en Java su uso es bastante sencillo.

Las colecciones en Java parten de una serie de interfaces básicas. Cada interfaz define un modelo de colección y las operaciones que se pueden llevar a cabo sobre los datos almacenados, por lo que es necesario conocerlas. La interfaz inicial, a través de la cual se han construido el resto de colecciones, es la interfaz `java.util.Collection`, que define las operaciones comunes a todas

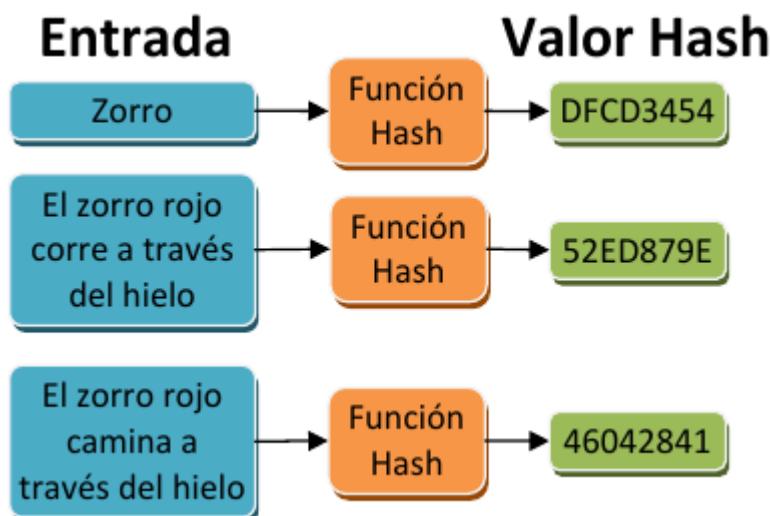
las colecciones derivadas. A continuación se muestran las operaciones más importantes definidas por esta interfaz, ten en cuenta que `Collection` es una interfaz genérica donde `<E>` es el parámetro de tipo (podría ser cualquier clase):

- **Método `int size()`** : retorna el número de elementos de la colección.
- **Método `boolean isEmpty()`** : retornará verdadero si la colección está vacía.
- **Método `boolean contains (Object element)`** : retornará verdadero si la colección tiene el elemento pasado como parámetro.
- **Método `boolean add(E element)`** : permitirá añadir elementos a la colección.
- **Método `boolean remove(Object element)`** : permitirá eliminar elementos de la colección.
- **Método `Iterator<E> iterator()`** : permitirá crear un iterador para recorrer los elementos de la colección. Esto se ve más adelante, no te preocupes.
- **Método `Object[] toArray()`** : permite pasar la colección a un array de objetos tipo Object.
- **Método `boolean containsAll(Collection<?> c)`** : permite comprobar si una colección contiene los elementos existentes en otra colección, si es así, retorna verdadero.
- **Método `boolean addAll(Collection<?> extends E> c)`** : permite añadir todos los elementos de una colección a otra colección, siempre que sean del mismo tipo (o deriven del mismo tipo base).
- **Método `boolean removeAll(Collection<?> c)`** : si los elementos de la colección pasada como parámetro están en nuestra colección, se eliminan, el resto se quedan.
- **Método `boolean retainAll(Collection<?> c)`** : si los elementos de la colección pasada como parámetro están en nuestra colección, se dejan, el resto se eliminan.
- **Método `void clear()`** : vaciar la colección.

Más adelante veremos cómo se usan estos métodos, será cuando veamos las implementaciones (clases genéricas que implementan alguna de las interfaces derivadas de la interfaz `Collection`).

24.4.2. Conjuntos (sets)

¿Con qué relacionarías los conjuntos? Seguro que con las matemáticas. Los conjuntos son un tipo de colección que no admite duplicados, derivados del concepto matemático de conjunto.



La interfaz `java.util.Set` define cómo deben ser los conjuntos, y implementa la interfaz `Collection`, aunque no añade ninguna operación nueva. Las implementaciones (clases genéricas que implementan la interfaz `Set`) más usadas son las siguientes:

- `java.util.HashSet`. Conjunto que almacena los objetos usando tablas hash (estructura de datos formada básicamente por un array donde la posición de los datos va determinada por una función hash, permitiendo localizar la información de forma extraordinariamente rápida. Los datos están ordenados en la tabla en base a un resumen numérico de los mismos (en hexadecimal generalmente) obtenido a partir de un algoritmo para cálculo de resúmenes, denominadas funciones hash. El resumen no tiene significado para un ser humano, se trata simplemente de un mecanismo para obtener un número asociado a un conjunto de datos. El inconveniente de estas tablas es que los datos se ordenan por el resumen obtenido, y no por el valor almacenado. El resumen, de un buen algoritmo hash, no se parece en nada al contenido almacenado) lo cual acelera enormemente el acceso a los objetos almacenados.

Inconvenientes: necesitan bastante memoria y no almacenan los objetos de forma ordenada (al contrario, pueden aparecer completamente desordenados).

- `java.util.LinkedHashSet`. Conjunto que almacena objetos combinando tablas hash, para un acceso rápido a los datos, y listas enlazadas (estructura de datos que almacena los objetos enlazándolos entre sí a través de un apuntador de memoria o puntero, manteniendo un orden, que generalmente es el del momento de inserción, pero que puede ser otro. Cada dato se almacena en una estructura llamada nodo en la que existe un campo, generalmente llamado siguiente, que contiene la dirección de memoria del siguiente nodo (con el siguiente dato)) para conservar el orden. El orden de almacenamiento es el de inserción, por lo que se puede decir que es una estructura ordenada a medias.
- **Inconvenientes:** necesitan bastante memoria y es algo más lenta que `HashSet`.
- `java.util.TreeSet`. Conjunto que almacena los objetos usando unas estructuras conocidas como árboles rojo-negro. Son más lentas que los dos tipos anteriores. pero tienen una gran ventaja: los datos almacenados se ordenan por valor. Es decir, que aunque se inserten los elementos de forma desordenada, internamente se ordenan dependiendo del valor de cada uno.

Poco a poco, iremos viendo que son las listas enlazadas y los árboles (no profundizaremos en los árboles rojo-negro, pero si veremos las estructuras tipo árbol en general). Veamos un ejemplo de uso básico de la estructura `HashSet` y después, profundizaremos en los `LinkedHashSet` y los `TreeSet`.

Para crear un conjunto, simplemente creamos el `HashSet` indicando el tipo de objeto que va a almacenar, dado que es una clase genérica que puede trabajar con cualquier tipo de dato debemos crearlo como sigue (no olvides hacer la importación de `java.util.HashSet` primero):

```
1 HashSet<Integer> conjunto=new HashSet<Integer>();
2 HashSet<Integer> conjunto=new HashSet<>(); //a partir de Java 7
```

Después podremos ir almacenando objetos dentro del conjunto usando el método `add` (definido por la interfaz `Set`). Los objetos que se pueden insertar serán siempre del tipo especificado al crear el conjunto:

```
1 Integer n=new Integer(10);
2 if (!conjunto.add(n)){
3     System.out.println("Número ya en la lista.");
4 }
```

Si el elemento ya está en el conjunto, el método `add` retornará `false` indicando que no se pueden insertar duplicados. Si todo va bien, retornará `true`.

24.4.2.1. ACCESO

Y ahora te preguntarás, ¿cómo accedo a los elementos almacenados en un conjunto? Para obtener los elementos almacenados en un conjunto hay que usar iteradores, que permiten obtener los elementos del conjunto uno a uno de forma secuencial (no hay otra forma de acceder a los elementos de un conjunto, es su inconveniente). Los iteradores se ven en mayor profundidad más adelante, de momento, vamos a usar iteradores de forma transparente, a través de una estructura `for` especial, denominada bucle "for-each" o bucle "para cada". En el siguiente código se usa un bucle `for-each`, en él la variable `i` va tomando todos los valores almacenados en el conjunto hasta que llega al último:

```
1 for (Integer i: conjunto) {
2     System.out.println("Elemento almacenado:"+i);
3 }
```

Como ves la estructura `for-each` es muy sencilla: la palabra `for` seguida de "(tipo variable:colección)" y el cuerpo del bucle; `tipo` es el tipo del objeto sobre el que se ha creado la colección, `variable` pues es la variable donde se almacenará cada elemento de la colección y `colección` la colección en sí. Los bucles `for-each` se pueden usar para todas las colecciones.

24.4.2.2. LinkedHashSet Y TreeSet

¿En qué se diferencian las estructuras `LinkedHashSet` y `TreeSet` de la estructura `HashSet`? Ya se comentó antes, y es básicamente en su funcionamiento interno.



La estructura `LinkedHashSet` es una estructura que internamente funciona como una lista enlazada, aunque usa también tablas hash para poder acceder rápidamente a los elementos. Una lista enlazada es una estructura similar a la representada en la imagen de la derecha, la cual está compuesta por nodos (elementos que forman la lista) que van enlazándose entre sí. Un nodo contiene dos cosas: el dato u objeto almacenado en la lista y el siguiente nodo de la lista. Si no hay siguiente nodo, se indica poniendo nulo (null) en la variable que contiene el siguiente nodo.

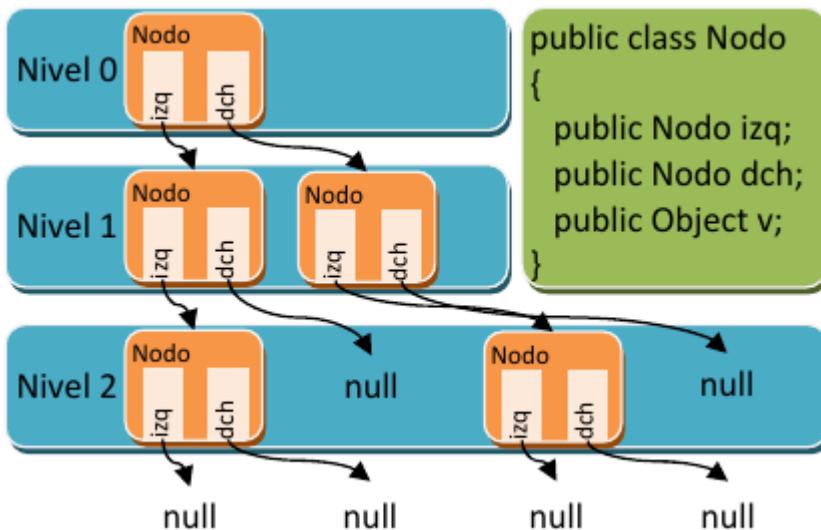
Las listas enlazadas tienen un montón de operaciones asociadas en las que no vamos a profundizar: eliminación de un nodo de la lista, inserción de un nodo al final, al principio o entre dos nodos, etc.

Gracias a las colecciones podremos utilizar listas enlazadas sin tener que complicarnos en detalles de programación.

La estructura `TreeSet`, en cambio, utiliza internamente árboles. Los árboles son como las listas pero mucho más complejos. En vez de tener un único elemento siguiente, pueden tener dos o más elementos siguientes, formando estructuras organizadas y jerárquicas.

Los nodos se diferencian en dos tipos: nodos padre y nodos hijo; un nodo padre puede tener varios nodos hijo asociados (depende del tipo de árbol), dando lugar a una estructura que parece un árbol invertido (de ahí su nombre).

En la figura de abajo se puede apreciar un árbol donde cada nodo puede tener dos hijos, denominados izquierdo (izq) y derecho (dch). Puesto que un nodo hijo puede también ser padre a su vez, los árboles se suelen visualizar para su estudio por niveles para entenderlos mejor, donde cada nivel contiene hijos de los nodos del nivel anterior, excepto el primer nivel (que no tiene padre).



Los árboles son estructuras complejas de manejar y que permiten operaciones muy sofisticadas. Los árboles usados en los `TreeSet`, los árboles rojo-negro, son árboles auto-ordenados, es decir, que al insertar un elemento, este queda ordenado por su valor de forma que al recorrer el árbol, pasando por todos los nodos, los elementos salen ordenados. El ejemplo mostrado en la imagen es simplemente un árbol binario, el más simple de todos.

Nuevamente, no se va a profundizar en las operaciones que se pueden realizar en un árbol a nivel interno (inserción de nodos, eliminación de nodos, búsqueda de un valor, etc.). Nos aprovecharemos de las colecciones para hacer uso de su potencial. En la siguiente tabla tienes un uso comparado de `TreeSet` y `LinkedHashSet`. Su creación es similar a como se hace con `HashSet`, simplemente sustituyendo el nombre de la clase `HashSet` por una de las otras. Ni `TreeSet`, ni `LinkedHashSet` admiten duplicados, y se usan los mismos métodos ya vistos antes, los existentes en la interfaz `Set` (que es la interfaz que implementan).

- Conjunto `TreeSet` ([Ejemplo01](#)):

```

1 TreeSet<Integer> t = new TreeSet<>();
2 t.add(4);
3 t.add(3);
4 t.add(1);
5 t.add(99);
6 for (Integer i : t) {
7     System.out.print(i + " ");
8 }

```

Resultado mostrado por pantalla (el resultado sale ordenado por valor):

```
1 1 3 4 99
```

- Conjunto `LinkedHashSet` ([Ejemplo02](#)):

```

1 LinkedHashSet<Integer> t = new LinkedHashSet<>();
2 t.add(4);
3 t.add(3);
4 t.add(1);
5 t.add(99);
6 for (Integer i : t) {
7     System.out.print(i + " ");
8 }

```

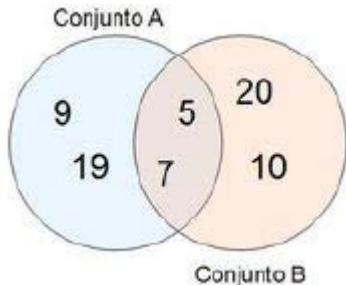
Resultado mostrado por pantalla (los valores salen ordenados según el momento de inserción en el conjunto):

```
1 4 3 1 99
```

24.4.2.3. OPERAR CON ELEMENTOS

¿Cómo podría copiar los elementos de un conjunto de uno a otro? ¿Hay que usar un bucle for y recorrer toda la lista para ello? ¡Qué va! Para facilitar esta tarea, los conjuntos, y las colecciones en general, facilitan un montón de operaciones para poder combinar los datos de varias colecciones. Ya se vieron en un apartado anterior, aquí simplemente vamos poner un ejemplo de su uso.

Partimos del siguiente ejemplo, en el que hay dos colecciones de diferente tipo, cada una con 4 números enteros:



```

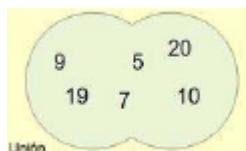
1 TreeSet<Integer> A= new TreeSet<Integer>();
2 A.add(9); A.add(19); A.add(5); A.add(7); // Elementos del conjunto A: 9, 19, 5 y 7
3 LinkedHashSet<Integer> B= new LinkedHashSet<Integer>();
4 B.add(10); B.add(20); B.add(5); B.add(7); // Elementos del conjunto B: 10, 20, 5 y 7

```

En el ejemplo anterior, el literal de número se convierte automáticamente a la clase envoltorio (wrapper) `Integer` sin tener que hacer nada, lo cual es una ventaja. Veamos las formas de combinar ambas colecciones:

- **Unión.** Añadir todos los elementos del conjunto B en el conjunto A.

```
1 A.addAll(B)
```

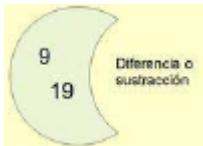


Todos los del conjunto A, añadiendo los del B, pero sin repetir los que ya están:

```
1  5, 7, 9, 10, 19 y 20.
```

- **Diferencia.** Eliminar los elementos del conjunto B que puedan estar en el conjunto A.

```
1  A.removeAll(B)
```

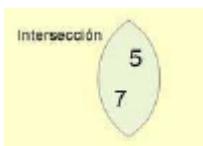


Todos los elementos del conjunto A, que no estén en el conjunto B:

```
1  9, 19.
```

- **Intersección.** Retiene los elementos comunes a ambos conjuntos.

```
1  A.retainAll(B)
```



Todos los elementos del conjunto A, que también están en el conjunto B:

```
1  5 y 7.
```

Recuerda

Estas operaciones son comunes a todas las colecciones.

Consulta el [Ejemplo03](#)

24.4.2.4. ORDENACIÓN

Por defecto, los `TreeSet` ordenan sus elementos de forma ascendente, pero, ¿se podría cambiar el orden de ordenación? Los `TreeSet` tienen un conjunto de operaciones adicionales, además de las que incluye por el hecho de ser un conjunto, que permite entre otras cosas, cambiar la forma de ordenar los elementos. Esto es especialmente útil cuando el tipo de objeto que se almacena no es un simple número, sino algo más complejo (un artículo por ejemplo). `TreeSet` es capaz de ordenar tipos básicos (números, cadenas y fechas) pero otro tipo de objetos no puede ordenarlos con tanta facilidad.

Para indicar a un `TreeSet` cómo tiene que ordenar los elementos, debemos decirle cuándo un elemento va antes o después que otro, y cuándo son iguales. Para ello, utilizamos la interfaz genérica `java.util.Comparator`, usada en general en algoritmos de ordenación, como veremos más adelante.

Se trata de crear una clase que implemente dicha interfaz, así de fácil. Dicha interfaz requiere de un único método que debe calcular si un objeto pasado por parámetro es mayor, menor o igual que otro del mismo tipo. Veamos un ejemplo general de cómo implementar un comparador para una hipotética clase `Objeto`:

```
1  class ComparadorDeObjetos implements Comparator<Objeto> {
2      public int compare(Objeto o1, Objeto o2) { ... }
3  }
```

La interfaz `Comparator` obliga a implementar un único método, es el método `compare`, el cual tiene dos parámetros: los dos elementos a comparar. Las reglas son sencillas, a la hora de personalizar dicho método:

- Si el primer objeto (o1) es menor que el segundo (o2), debe retornar un número entero negativo.
- Si el primer objeto (o1) es mayor que el segundo (o2), debe retornar un número entero positivo.
- Si ambos son iguales, debe retornar 0.

A veces, cuando el orden que deben tener los elementos es diferente al orden real (por ejemplo cuando ordenamos los números en orden inverso), la definición de antes puede ser un poco liosa, así que es recomendable en tales casos pensar de la siguiente forma:

- Si el primer objeto (o1) debe ir antes que el segundo objeto (o2), retornar entero negativo.
- Si el primer objeto (o1) debe ir después que el segundo objeto (o2), retornar entero positivo.
- Si ambos son iguales, debe retornar 0.

Una vez creado el comparador simplemente tenemos que pasarlo como parámetro en el momento de la creación al `TreeSet`, y los datos internamente mantendrán dicha ordenación:

```
1 TreeSet<Objeto> ts=new TreeSet<>(new ComparadorDeObjetos());
```

Hay otra manera de definir esta ordenación, pero lo estudiaremos más a fondo en el punto [Comparadores](#)

Para entender mejor los Sets revisa el [Ejemplo04](#)

24.4.3. Listas

¿En qué se diferencia una lista de un conjunto? Las listas son elementos de programación un poco más avanzados que los conjuntos. Su ventaja es que amplían el conjunto de operaciones de las colecciones añadiendo operaciones extra, veamos algunas de ellas:

- Las listas si **pueden almacenar duplicados**, si no queremos duplicados, hay que verificar manualmente que el elemento no esté en la lista antes de su inserción.
- **Acceso posicional.** Podemos acceder a un elemento indicando su posición en la lista.
- **Búsqueda.** Es posible buscar elementos en la lista y obtener su posición. En los conjuntos, al ser colecciones sin aportar nada nuevo, solo se podía comprobar si un conjunto contenía o no un elemento de la lista, retornando verdadero o falso. Las listas mejoran este aspecto.
- **Extracción de sublistas.** Es posible obtener una lista que contenga solo una parte de los elementos de forma muy sencilla.

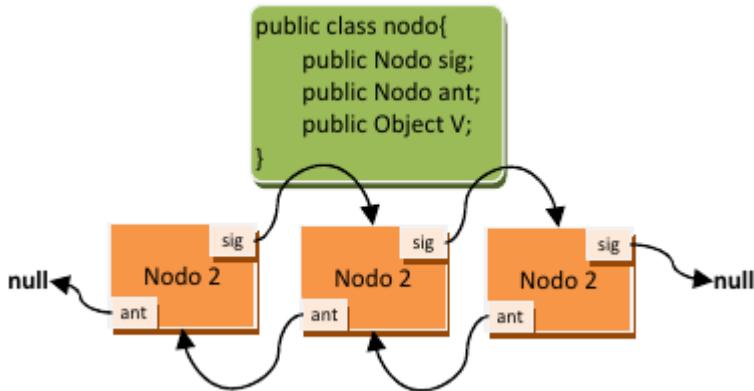
En Java, para las listas se dispone de una interfaz llamada `java.util.List`, y dos implementaciones (`java.util.LinkedList` y `java.util.ArrayList`), con diferencias significativas entre ellas. Los métodos de la interfaz `List`, que obviamente estarán en todas las implementaciones, y que permiten las operaciones anteriores son:

- `E get(int index)`. El método `get` permite obtener un elemento partiendo de su posición (`index`).
- `E set(int index, E element)`. El método `set` permite cambiar el elemento almacenado en una posición de la lista (`index`), por otro (`element`).
- `void add(int index, E element)`. Se añade otra versión del método `add`, en la cual se puede insertar un elemento (`element`) en la lista en una posición concreta (`index`), desplazando los existentes.
- `E remove(int index)`. Se añade otra versión del método `remove`, esta versión permite eliminar un elemento indicando su posición en la lista.
- `boolean addAll(int index, Collection<? extends E> c)`. Se añade otra versión del método `addAll`, que permite insertar una colección pasada por parámetro en una posición de la lista, desplazando el resto de elementos.
- `int indexOf(Object o)`. El método `indexOf` permite conocer la posición (índice) de un elemento, si dicho elemento no está en la lista retornará `-1`.
- `int lastIndexOf(Object o)`. El método `lastIndexOf` nos permite obtener la última ocurrencia del objeto en la lista (dado que la lista si puede almacenar duplicados).
- `List<E> subList(int from, int to)`. El método `subList` genera una sublista (una vista parcial de la lista) con los elementos comprendidos entre la posición inicial (incluida) y la posición final (no incluida).

Ten en cuenta que los elementos de una lista empiezan a numerarse por 0. Es decir, que el primer elemento de la lista es el 0. Ten en cuenta también que `List` es una interfaz genérica, por lo que `<E>` corresponde con el tipo base usado como parámetro genérico al crear la lista.

24.4.3.1. USO

Y, ¿cómo se usan las listas? Pues para usar una lista haremos uso de sus implementaciones `LinkedList` y `ArrayList`. Veamos un ejemplo de su uso y después obtendrás respuesta a esta pregunta.



Supongo que intuirás como se usan, pero nunca viene mal un ejemplo sencillo, que nos aclare las ideas. El siguiente ejemplo muestra como usar un `LinkedList` pero valdría también para `ArrayList` (no olvides importar las clases `java.util.LinkedList` y `java.util.ArrayList` según sea necesario). En este ejemplo se usan los métodos de acceso posicional a la lista:

```

1  LinkedList<Integer> t=new LinkedList<>(); // Declaración y creación del LinkedList de enteros.
2  t.add(1); // Añade un elemento al final de la lista.
3  t.add(3); // Añade otro elemento al final de la lista.
4  t.add(1,2); // Añade en la posición 1 el elemento 2.
5  t.add(t.get(1)+t.get(2)); // Suma los valores contenidos en la posición 1 y 2, y lo agrega al final.
6  t.remove(0); // Elimina el primer elementos de la lista.

```

En el ejemplo anterior, se realizan muchas operaciones, ¿cuál será el contenido de la lista al final? Pues será 2, 3 y 5. En el ejemplo cabe destacar el uso del bucle `for-each`, recuerda que se puede usar en cualquier colección.

Veamos otro ejemplo, esta vez con `ArrayList`, de cómo obtener la posición de un elemento en la lista:

```

1  ArrayList<Integer> al=new ArrayList<>(); // Declaración y creación del ArrayList de enteros.
2  al.add(10); al.add(11); // Añadimos dos elementos a la lista.
3  al.set(al.indexOf(11), 12); // Sustituimos el 11 por el 12, primero lo buscamos y luego lo reemplazamos.

```

En el ejemplo anterior, se emplea tanto el método `indexOf` para obtener la posición de un elemento, como el método `set` para reemplazar el valor en una posición, una combinación muy habitual. El ejemplo anterior generará un `ArrayList` que contendrá dos números, el 10 y el 12. Veamos ahora un ejemplo algo más difícil:

```

1  al.addAll(0, t.subList(1, t.size()));

```

Información

`subList` ➔ Returns a view of the portion of this list between the specified `fromIndex`, inclusive, and `toIndex`, exclusive. ([API de Java](#))

Este ejemplo es especial porque usa sublistas. Se usa el método `size` para obtener el tamaño de la lista. Después el método `subList` para extraer una sublista de la lista (que incluía en origen los números 2, 3 y 5), desde la posición 1 hasta el final de la lista (lo cual dejaría fuera al primer elemento). Y por último, se usa el método `addAll` para añadir todos los elementos de la sublista al `ArrayList` anterior. Y quedaria:

```

1  3, 5, 10 y 12.

```

Debes saber que las operaciones aplicadas a una sublista repercuten sobre la lista original. Por ejemplo, si ejecutamos el método `clear` sobre una sublista, se borrarán todos los elementos de la sublista, pero también se borrarán dichos elementos de la lista original:

```

1  ArrayList<Integer> alAux=new ArrayList<>();
2  alAux = al.subList(0, 2);
3  alAux.clear();
4
5  al.subList(0, 2).clear();

```

Lo mismo ocurre al añadir un elemento, se añade en la sublista y en la lista original.

Puedes consultar el código en el [Ejemplo05](#)

24.4.3.2. `LinkedList` Y `ArrayList`

¿Y en qué se diferencia un `LinkedList` de un `ArrayList` ?

Los `LinkedList` utilizan listas doblemente enlazadas, que son listas enlazadas (como se vio en un apartado anterior), pero que permiten ir hacia atrás en la lista de elementos. Los elementos de la lista se encapsulan en los llamados nodos.

Los nodos van enlazados unos a otros para no perder el orden y no limitar el tamaño de almacenamiento. Tener un doble enlace significa que en cada nodo se almacena la información de cuál es el siguiente nodo y además, de cuál es el nodo anterior. Si un nodo no tiene nodo siguiente o nodo anterior, se almacena null o nulo para ambos casos.

No es el caso de los `ArrayList`. Estos se implementan utilizando arrays que se van redimensionando conforme se necesita más espacio o menos. La redimensión es transparente a nosotros, no nos enteramos cuando se produce, pero eso redonda en una diferencia de rendimiento notable dependiendo del uso. Los `ArrayList` son más rápidos en cuanto a acceso a los elementos, acceder a un elemento según su posición es más rápido en un array que en una lista doblemente enlazada (hay que recorrer la lista). En cambio, eliminar un elemento implica muchas más operaciones en un array que en una lista enlazada de cualquier tipo.

¿Y esto que quiere decir? Que si se van a realizar muchas operaciones de eliminación de elementos sobre la lista, conviene usar una lista enlazada (`LinkedList`), pero si no se van a realizar muchas eliminaciones, sino que solamente se van a insertar y consultar elementos por posición, conviene usar una lista basada en arrays redimensionados (`ArrayList`).

`LinkedList` tiene otras ventajas que nos puede llevar a su uso. Implementa las interfaces `java.util.Queue` y `java.util.Deque`. Dichas interfaces permiten hacer uso de las listas como si fueran una cola de prioridad o una pila, respectivamente.

Las colas, también conocidas como colas de prioridad, son una lista pero que aportan métodos para trabajar de forma diferente. ¿Tú sabes lo que es hacer cola para que te atiendan en una ventanilla?

Pues igual. Se trata de que el que primero que llega es el primero en ser atendido (FIFO en inglés). Simplemente se aportan tres métodos nuevos: meter en el final de la lista (`add` y `offer`), sacar y eliminar el elemento más antiguo (`poll`), y examinar el elemento al principio de la lista sin eliminarlo (`peek`). Dichos métodos están disponibles en las listas enlazadas `LinkedList`:

- `boolean add(E e)` y `boolean offer(E e)`, retornarán true si se ha podido insertar el elemento al final de la `LinkedList`.
- `E poll()` retornará el primer elemento de la `LinkedList` y lo eliminará de la misma. Al insertar al final, los elementos más antiguos siempre están al principio. Retornará null si la lista está vacía.
- `E peek()` retornará el primer elemento de la `LinkedList` pero no lo eliminará, permite examinarlo. Retornará null si la lista está vacía.

Las pilas, mucho menos usadas, son todo lo contrario a las listas. Una pila es igual que una montaña de hojas en blanco, para añadir hojas nuevas se ponen encima del resto, y para retirar una se coge la primera que hay, encima de todas. En las pilas el último en llegar es el primero en ser atendido. Para ello se proveen de tres métodos: meter al principio de la pila (`push`), sacar y eliminar del principio de la pila (`pop`), y examinar el primer elemento de la pila (`peek`, igual que si usara la lista como una cola). Las pilas se usan menos y haremos menos hincapié en ellas. Simplemente ten en mente que, tanto las colas como las pilas, son una lista enlazada sobre la que se hacen operaciones especiales.

24.4.3.3. A TENER EN CUENTA

A la hora de usar las listas, hay que tener en cuenta un par de detalles, ¿sabes cuáles? Es sencillo, pero importante.

No es lo mismo usar las colecciones (listas y conjuntos) con objetos inmutables (`String`, `Integer`, etc.) que con objetos mutables. Los objetos inmutables no pueden ser modificados después de su creación, por lo que cuando se incorporan a la lista, a través de los métodos `add`, se pasan por copia (es decir, se realiza una copia de los mismos). En cambio los objetos mutables (como las clases que tú puedes crear), no se copian, y eso puede producir efectos no deseados.

Imagínate la siguiente clase, que contiene un número:

```

1  class Test {
2      public Integer num;
3      Test (int num) {
4          this.num=new Integer(num);
5      }
6  }
```

La clase de antes es mutable, por lo que no se pasa por copia a la lista. Ahora imagina el siguiente código en el que se crea una lista que usa este tipo de objeto, y en el que se insertan dos objetos:

```

1 Test p1=new Test(11); // Se crea un objeto Test donde el entero que contiene vale 11.
2 Test p2=new Test(12); // Se crea otro objeto Test donde el entero que contiene vale 12.
3 LinkedList<Test> lista=new LinkedList<Test>(); // Creamos una lista enlazada para objetos tipo Test.
4 lista.add(p1); // Añadimos el primero objeto test.
5 lista.add(p2); // Añadimos el segundo objeto test.
6 for (Test p:lista){
7     System.out.println(p.num); // Mostramos la lista de objetos.
8 }

```

¿Qué mostraría por pantalla el código anterior? Simplemente mostraría los números 11 y 12. Ahora bien, ¿qué pasa si modificamos el valor de uno de los números de los objetos test? ¿Qué se mostrará al ejecutar el siguiente código?

```

1 p1.num=44;
2 for (Test p:lista) System.out.println(p.num);

```

El resultado de ejecutar el código anterior es que se muestran los números 44 y 12. El número ha sido modificado y no hemos tenido que volver a insertar el elemento en la lista para que en la lista se cambie también. Esto es porque en la lista no se almacena una copia del objeto Test, sino un apuntador a dicho objeto (solo hay una copia del objeto a la que se hace referencia desde distintos lugares).

Información

"Controlar la complejidad es la esencia de la programación." **Brian Kernighan**

Consulta el [Ejemplo06](#)

24.4.4. Conjuntos de pares [clave/valor] (Diccionario)

¿Cómo almacenarías los datos de un diccionario? Tenemos por un lado cada palabra y por otro su significado. Para resolver este problema existen precisamente los arrays asociativos. Un tipo de array asociativo son los mapas o diccionarios, que permiten almacenar pares de valores conocidos como clave y valor. La clave se utiliza para acceder al valor, como una entrada de un diccionario permite acceder a su definición.

En Java existe la interfaz `java.util.Map` que define los métodos que deben tener los mapas, y existen tres implementaciones principales de dicha interfaz: `java.util.HashMap`, `java.util.TreeMap` y `java.util.LinkedHashMap`. ¿Te suenan? Claro que sí. Cada una de ellas, respectivamente, tiene características similares a `HashSet`, `TreeSet` y `LinkedHashSet`, tanto en funcionamiento interno como en rendimiento.

Los mapas utilizan clases genéricas para dar extensibilidad y flexibilidad, y permiten definir un tipo base para la clave, y otro tipo diferente para el valor. Veamos un ejemplo de como crear un mapa, que es extensible a los otros dos tipos de mapas:

```
1 HashMap<String, Integer> t = new HashMap<>();
```

El mapa anterior permite usar cadenas como llaves y almacenar de forma asociada a cada llave, un número entero. Veamos los métodos principales de la interfaz `Map`, disponibles en todas las implementaciones. En los ejemplos, `V` es el tipo base usado para el valor (`Value`) y `K` el tipo base usado para la llave (`Key`):

Método.	Descripción.
<code>V put(K key, V value);</code>	Inserta un par de objetos llave (key) y valor (value) en el mapa. Si la llave ya existe en el mapa, entonces retornará el valor asociado que tenía antes, si la llave no existía, entonces retornará null.
<code>V get(Object key);</code>	Obtiene el valor asociado a una llave ya almacenada en el mapa. Si no existe la llave, retornará null.
<code>V remove(Object key);</code>	Elimina la llave y el valor asociado. Retorna el valor asociado a la llave, por si lo queremos utilizar para algo, o null, si la llave no existe.
<code>boolean containsKey(Object key);</code>	Retornará true si el mapa tiene almacenada la llave pasada por parámetro, false en cualquier otro caso.
<code>boolean containsValue(Object value);</code>	Retornará true si el mapa tiene almacenado el valor pasado por parámetro, false en cualquier otro caso.
<code>int size();</code>	Retornará el número de pares llave y valor almacenado en el mapa.

Método.	Descripción.
<code>boolean isEmpty();</code>	Retornará true si el mapa está vacío, false en cualquier otro caso.
<code>void clear();</code>	Vacía el mapa.

Revisa el [Ejemplo07](#)

24.4.4.1. RECORRIDO CON `keySet` O `entrySet`

```

1 // Recorrido por claves
2 for (String clave:divisas.keySet()){
3     System.out.format("%s --> %.02f\n", clave, divisas.get(clave));
4 }
5 // Recomendado cuando solo necesito las claves (aunque a partir de la clave puedes recuperar el valor)
6
7 // Recorrido por entries (pares clave/valor)
8 for (Map.Entry<String, Double> entry : divisas.entrySet()){
9     System.out.format("%s --> %.02f\n", entry.getKey(), entry.getValue());
10 }
11 // Recomendado cuando necesito la clave y el valor

```

24.5. Iteradores

¿Qué son los iteradores realmente? Son un mecanismo que nos permite recorrer todos los elementos de una colección de forma sencilla, de forma secuencial, y de forma segura. Los mapas, como no derivan de la interfaz `Collection` realmente, no tienen iteradores, pero como veremos, existe un truco interesante.

Los iteradores permiten recorrer las colecciones de dos formas: bucles `for-each` (existentes en Java a partir de la versión 1.5) y a través de un bucle normal creando un iterador. Como los bucles `for-each` ya los hemos visto antes (y ha quedado patente su simplicidad), nos vamos a centrar en el otro método, especialmente útil en versiones antiguas de Java. Ahora la pregunta es, ¿cómo se crea un iterador? Pues invocando el método "`iterator()`" de cualquier colección.

Veamos un ejemplo (en el ejemplo `t` es una colección cualquiera):

```
1 Iterator<Integer> it=t.iterator();
```

Fíjate que se ha especificado un parámetro para el tipo de dato genérico en el iterador (poniendo `<Integer>` después de `Iterator`). Esto es porque los iteradores son también clases genéricas, y es necesario especificar el tipo base que contendrá el iterador. Si no se especifica el tipo base del iterador, igualmente nos permitiría recorrer la colección, pero retornaría objetos tipo `Object` (clase de la que derivan todas las clases), con lo que nos veremos obligados a forzar la conversión de tipo.

Para recorrer y gestionar la colección, el iterador ofrece tres métodos básicos:

- `boolean hasNext()`. Retornará true si le quedan más elementos a la colección por visitar. False en caso contrario.
- `E next()`. Retornará el siguiente elemento de la colección, si no existe siguiente elemento, lanzará una excepción (`NoSuchElementException` para ser exactos), con lo que conviene chequear primero si el siguiente elemento existe.
- `remove()`. Elimina de la colección el último elemento retornado en la última invocación de `next` (no es necesario pasarselo por parámetro). Cuidado, si `next` no ha sido invocado todavía, saltará una incomoda excepción.

¿Cómo recorreríamos una colección con estos métodos? Pues de una forma muy sencilla, un simple bucle mientras (`while`) con la condición `hasNext()` nos permite hacerlo:

```

1 while (it.hasNext()) // Mientras haya siguiente elemento, seguiremos en el bucle.
2 {
3     Integer n=it.next(); // Escogemos el siguiente elemento.
4     if (n%2==0) it.remove(); //Si es par, eliminamos el elemento de la lista.
5 }
```

¿Qué elementos contendría la lista después de ejecutar el bucle? Efectivamente, solo los números impares.

Información

Las listas permiten acceso posicional a través de los métodos `get` y `set`, y acceso secuencial a través de iteradores, ¿cuál es para tí la forma más cómoda de recorrer todos los elementos? ¿Un acceso posicional a través un bucle `for (i=0;i<lista.size();i++)` o un acceso secuencial usando un bucle `while (iterador.hasNext())`?

¿Qué inconvenientes tiene usar los iteradores sin especificar el tipo de objeto? En el siguiente ejemplo, se genera una lista con los números del 0 al 10. De la lista, se eliminan aquellos que son pares y solo se dejan los impares. En el primer ejemplo se especifica el tipo de objeto del iterador, en el segundo ejemplo no, observa el uso de la conversión de tipos en la línea 7.

Ejemplo indicando el tipo de objeto de iterador.

```

1 ArrayList<Integer> lista=new ArrayList<>();
2 for (int i=0;i<10;i++){
3     lista.add(i);
4 }
5 //lista: [0,1,2,3,4,5,6,7,8,9]
6 Iterator<Integer> it=lista.iterator();
7 while (it.hasNext()) {
8     Integer n=it.next();
9     if (n%2==0){
10         it.remove();
11     }
12 }
13 //lista: [1,3,5,7,9]
```

Ejemplo no indicando el tipo de objeto del iterador,

```

1 ArrayList <Integer> lista=new ArrayList<Integer>();
2 for (int i=0;i<10;i++){
3     lista.add(i);
4 }
5 Iterator it=lista.iterator();
6 while (it.hasNext()) {
7     Integer n=(Integer)it.next();
8     if (n%2==0){
9         it.remove();
10    }
11 }
```

Un iterador es seguro porque esta pensado para no sobrepasar los límites de la colección, ocultando operaciones más complicadas que pueden repercutir en errores de software. Pero realmente se convierte en inseguro cuando es necesario hacer la operación de conversión de tipos. Si la colección no contiene los objetos esperados, al intentar hacer la conversión, saltará una incómoda excepción.

Usar genéricos aporta grandes ventajas, pero usándolos adecuadamente.

Para recorrer los mapas con iteradores, hay que hacer un pequeño truco. Usamos el método `entrySet` que ofrecen los mapas para generar un conjunto con las entradas (pares de llave-valor), o bien, el método `keySet` para generar un conjunto con las llaves existentes en el mapa. Veamos como sería para el segundo caso, el más sencillo:

```

1 HashMap<Integer,Integer> mapa=new HashMap<>();
2 for (int i=0;i<10;i++){
3     mapa.put(i, i); // Insertamos datos de prueba en el mapa.
4 }
5 for (Integer llave:mapa.keySet()){
6     // Recorremos el conjunto generado por keySet, contendrá las llaves.
7     Integer valor=mapa.get(llave); //Para cada llave, accedemos a su valor si es necesario.
8 }
```

Lo único que tienes que tener en cuenta es que el conjunto generado por `keySet` no tendrá obviamente el método `add` para añadir elementos al mismo, dado que eso tendrás que hacerlo a través del mapa.

Acción

Si usas iteradores, y piensas eliminar elementos de la colección (e incluso de un mapa), debes usar el método `remove` del iterador y no el de la colección. Si eliminas los elementos utilizando el método `remove` de la colección, mientras estás dentro de un bucle de iteración, o dentro de un bucle `for-each`, los fallos que pueden producirse en tu programa son impredecibles. ¿Logras adivinar porqué se pueden producir dichos problemas?

Los problemas son debidos a que el método remove del iterador elimina el elemento de dos sitios: de la colección y del iterador en sí (que mantiene interiormente información del orden de los elementos). Si usas el método remove de la colección, la información solo se elimina de un lugar, de la colección.

Consulta el [Ejemplo08](#) y el [Ejemplo09](#) (que es la versión del [Ejemplo06](#) con iteradores).

24.6. Comparadores

En Java hay dos mecanismos para cambiar la forma en la que los elementos se ordenan. Imagina que tienes los artículos almacenados en una lista llamada `articulos`, y que cada artículo se almacena en la siguiente clase `Articulo` (fíjate que el código de artículo es una cadena y no un número):

```
1 class Articulo {
2     public String codArticulo; // Código de artículo
3     public String descripcion; // Descripción del artículo.
4     public int cantidad; // Cantidad a proveer del artículo.
5 }
```

La primera forma de ordenar consiste en crear una clase que implemente la interfaz `java.util.Comparator`, y por ende, el método `compare` definido en dicha interfaz. Esto se explicó en el apartado de conjuntos, al explicar el `TreeSet`, así que no vamos a profundizar en ello. No obstante, el comparador para ese caso podría ser así:

```
1 class comparadorArticulos implements Comparator<Articulo>{
2     @Override
3     public int compare( Articulo o1, Articulo o2 ) {
4         return o1.codArticulo.compareTo(o2.codArticulo);
5     }
6 }
```

Una vez creada esta clase, ordenar los elementos es muy sencillo, simplemente se pasa como segundo parámetro del método `sort` una instancia del comparador creado:

```
1 Collections.sort(coleccionArticulos, new comparadorArticulos());
```

La segunda forma es quizás más sencilla cuando se trata de objetos cuya ordenación no existe de forma natural, pero requiere modificar la clase `Articulo`. Consiste en hacer que los objetos que se meten en la lista o array implementen la interfaz `java.util.Comparable`. Todos los objetos que implementan la interfaz `Comparable` son "ordenables" y se puede invocar el método `sort` sin indicar un comparador para ordenarlos. La interfaz `comparable` solo requiere implementar el método `compareTo`:

```
1 class Articulo implements Comparable<Articulo>{
2     public String codArticulo;
3     public String descripcion;
4     public int cantidad;
5
6     @Override
7     public int compareTo(Articulo o) {
8         return codArticulo.compareTo(o.codArticulo);
9     }
10 }
```

Del ejemplo anterior se pueden denotar dos cosas importantes: que la interfaz `Comparable` es genérica y que para que funcione sin problemas es conveniente indicar el tipo base sobre el que se permite la comparación (en este caso, el objeto `Articulo` debe compararse consigo mismo), y que el método `compareTo` solo admite un parámetro, dado que comparará el objeto con el que se pasa por parámetro.

El funcionamiento del método `compareTo` es el mismo que el método `compare` de la interfaz `Comparator`: si la clase que se pasa por parámetro es igual al objeto, se tendría que retornar 0; si es menor o anterior, se debería retornar un número menor que cero; si es mayor o posterior, se debería retornar un número mayor que 0.

Ordenar ahora la lista de artículos es sencillo, fíjate que fácil: `Collections.sort(coleccionArticulos);`

Consulta el código de [Ejemplo10](#) y [Ejemplo11](#)

24.7. Extras

¿Qué más ofrece las clases `java.util.Collections` y `java.util.Arrays` de Java? Una vez vista la ordenación, quizás lo más complicado, veamos algunas operaciones adicionales. En los ejemplos, la variable `array` es un array y la variable `lista` es una lista de cualquier tipo de elemento:

Operación	Descripción	Ejemplos
Desordenar una lista.	Desordena una lista, este método no está disponible para arrays.	<code>Collections.shuffle (lista);</code>

Operación	Descripción	Ejemplos
Rellenar una lista o array.	Rellena una lista o array copiando el mismo valor en todos los elementos del array o lista. Útil para reiniciar una lista o array.	Collections.fill (lista,elemento); Arrays.fill (array,elemento);
Búsqueda binaria.	Permite realizar búsquedas rápidas en una lista o array ordenados. Es necesario que la lista o array estén ordenados, sino lo están, la búsqueda no tendrá éxito.	Collections.binarySearch(lista,elemento); Arrays.binarySearch(array, elemento);
Convertir un array a lista.	Permite rápidamente convertir un array a una lista de elementos, extremadamente útil. No se especifica el tipo de lista retornado (no es ArrayList ni LinkedList), solo se especifica que retorna una lista que implementa la interfaz java.util.List.	List lista = Arrays.asList(array); Si el tipo de dato almacenado en el array es conocido (Integer por ejemplo), es conveniente especificar el tipo de objeto de la lista: List<Integer> lista = Arrays.asList(array);
Convertir una lista a array.	Permite convertir una lista en array. Esto se puede realizar en todas las colecciones, y no es un método de la clase Collections, sino propio de la interfaz collection. Es conveniente que sepas de su existencia.	Para este ejemplo, supondremos que los elementos de la lista son números, dado que hay que crear un array del tipo almacenado en la lista, y del tamaño de la lista: Integer[] array=new Integer[lista.size()]; lista.toArray(array);
Dar la vuelta.	Da la vuelta a una lista, poniéndola en orden inverso al que tiene.	Collections.reverse(lista);
Imprimir un array o lista		lista.toString() Arrays.toString(array)

Otra operación que ya se ha visto en algún ejemplo anterior es la de dividir una cadena en partes. Cuando una cadena está formada internamente por trozos de texto claramente delimitados por un separador (una coma, un punto y coma o cualquier otro), es posible dividir la cadena y obtener cada uno de los trozos de texto por separado en un array de cadenas.

Para poder realizar esta operación, usaremos el método `split` de la clase `String`. El delimitador o separador es una expresión regular, único argumento del método `split`, y puede ser obviamente todo lo complejo que sea necesario:

```

1 String texto="Z,B,A,X,M,O,P,U";
2 String[] partes=texto.split(",");
3 //partes={"Z", "B", "A", "X", "M", "O", "P", "U"}
4 Arrays.sort(partes);//lo ordenamos
5 //partes={"A", "B", "M", "O", "P", "U", "X", "Z"} 
```

En el ejemplo anterior la cadena `texto` contiene una serie de letras separadas por comas. La cadena se ha dividido con el método `split`, y se ha guardado cada carácter por separado en un `array`. Después se ha ordenado el `array`. ¡Increíble lo que se puede llegar a hacer con solo tres líneas de código!

24.8. Programación funcional

24.8.1. ¿Qué es la programación funcional?

Paradigma de programación **declarativo**, no imperativo, se dice cómo es el problema a resolver, en lugar de los pasos a seguir para resolverlo.

La mayoría de lenguajes populares actuales no se pueden considerar funcionales, ni puros ni híbridos, pero han adaptado su sintaxis y funcionalidad para ofrecer parte de este paradigma.

24.8.2. Características principales

Transparencia referencial: la salida de una función debe depender sólo de sus argumentos. Si la llamamos varias veces con los mismos argumentos, debe producir siempre el mismo resultado.

Inmutabilidad de los datos: los datos deben ser inmutables para evitar posibles efectos colaterales.

Composición de funciones: las funciones se tratan como datos, de modo que la salida de una función se puede tomar como entrada para la siguiente.

Funciones de primer orden: funciones que permiten tener otras funciones como parámetros, a modo de callbacks.

Información

Si llamamos repetidamente a esta función con el parámetro 1, cada vez producirá un resultado distinto (3, 4, 5...)

```

1 class Prueba{
2     static int valorExterno = 1;
3
4     static int unaFuncion(int parametro){
5         valorExterno++;
6         return valorExterno + parametro;
7     }
8 }
```

Imperativo vs Declarativo

Queremos obtener una sublista con los mayores de edad de entre una lista de personas:

Imperativo:

```

1 List<Persona> adultos = new ArrayList<>();
2 for (int i = 0; i < personas.size(); i++){
3     if (personas.get(i).getEdad() >= 18)
4         adultos.add(personas.get(i));
5 }
```

Declarativo

```
1 List<Persona> adultos = personas.stream().filter(p -> p.getEdad() >= 18).collect(Collectors.toList());
```

Se puede observar que el ejemplo declarativo es más compacto, y menos propenso a errores. (Además sirve de ejemplo a la composición de funciones)

24.8.3. Funciones Lambda

Son expresiones breves que simplifican la implementación de elementos más costosos en cuanto a líneas de código. También se las conoce como funciones anónimas, no necesitan una clase/nombre. En java se pueden aplicar a la implementación de interfaces, aunque tienen más utilidades prácticas. En algunos lenguajes se les suele denominar "funciones flecha" (arrow functions) ya que en su sintaxis es característica una flecha, que separa la cabecera de la función de su cuerpo.

Comparaciones

API del método `List.sort` de Java:

```
1 default void sort(Comparator<? super E> c)
```

La interfaz `Comparator` pide implementar un método `compare`, que recibe dos datos del tipo a tratar (`T`), y devuelve un entero indicando si el primero es menor, mayor, o son iguales (de forma similar al método `compareTo` de la interfaz `Comparable`).

```
1 int compare (T o1, T o2)
```

Imaginemos una clase `Persona`:

```
1 class Persona{
2     private String nombre;
3     private int edad;
4     ...
5 }
```

Y un `ArrayList` `personas` formada por objetos de tipo `Persona`:

```
1 ...
2 ArrayList<Persona> personas = new ArrayList<>();
3 personas.add(new Persona("Nacho", 52));
4 personas.add(new Persona("David", 47));
5 personas.add(new Persona("Pepe", 42));
6 personas.add(new Persona("Maria", 22));
7 personas.add(new Persona("Marta", 4));
8 ...
```

Ahora queremos ordenar el `ArrayList` de `personas` de mayor a menor edad usando...

Implementación "tradicional" java: `Comparator` o `Comparable`

```
1 ...
2     class ComparadorPersona implements Comparator <Persona>{
3         @Override
4         public int compare(Persona p1, Persona p2){
5             return p2.getEdad() - p1.getEdad();
6         }
7     }
8 ...
```

```
1 ...
2 personas.sort(new ComparadorPersona());
3 for (int i = 0; i < personas.size(); i++){
4     System.out.println(personas.get(i));
5 }
6 ...
```

Sin embargo, implementado con funciones Lambda sería...

```
1 ...
2 personas.sort((p1, p2) -> p2.getEdad() - p1.getEdad());
3 for (int i = 0; i < personas.size(); i++){
4     System.out.println(personas.get(i));
5 }
6 ...
```

24.8.3.1. ESTRUCTURA DE UNA EXPRESIÓN LAMBDA

```
(lista de parámetros) -> {cuerpo de la función a implementar}
```

- El **operador lambda** (`->`) separa la declaración de parámetros de la declaración del cuerpo de la función.
- Los **parámetros** del lado izquierdo de la flecha se pueden omitir si sólo hay un parámetro. Cuando no se tienen parámetros, o cuando se tienen dos o más, es necesario utilizar paréntesis.
- El **cuerpo de la función** son las llaves de la parte derecha se pueden omitir si la única operación a realizar es un simple `return`.

24.8.3.2. FUNCIONES LAMBDA (LAS UTILIZAREMOS A FONDO CON LAS Interfaces)

```
1 z -> z + 2 //un sólo parámetro
```

```

1 () -> System.out.println("Mensaje 1") //sin parámetros

1 (int longitud, int altura) -> { return altura * longitud; } //dos parámetros

1 (String x) -> {
2   String retorno = x;
3   retorno = retorno.concat("****");
4   return retorno;
5 } //un bloque de código más elaborado

```

24.8.4. Gestión de colecciones con streams en Java

Desde Java 8, permiten procesar grandes cantidades de datos aprovechando la paralelización que permite el sistema. No modifican la colección original, sino que crean copias.

Dos tipos de operaciones

- **Intermedias:** devuelven otro stream resultado de procesar el anterior de algún modo (filtrado, mapeo), para ir enlazando operaciones
- **Finales:** cierran el stream devolviendo algún resultado (colección resultante, cálculo numérico, etc).

Muchas de estas operaciones tienen como parámetro una interfaz, que puede implementarse muy brevemente empleando expresiones lambda

24.8.4.1. FILTRADO

El método `filter` es una operación intermedia que permite quedarnos con los datos de una colección que cumplan el criterio indicado como parámetro. `filter` recibe como parámetro una interfaz `Predicate`, cuyo método `test` recibe como parámetro un objeto y devuelve si ese objeto cumple o no una determinada condición.

```

1 [...]
2 Stream<Persona> adultos = personas.stream().filter(p -> p.getEdad() >= 18);
3 //La función lambda se podría traducir como: "Aquellas personas 'p' de la colección cuya edad sea mayor o igual que 18 años"

```

24.8.4.2. MAPEO

El método `map` es una operación intermedia que permite transformar la colección original para quedarnos con cierta parte de la información o crear otros datos. `map` recibe como parámetro una interfaz `Function`, cuyo método `apply` recibe como parámetro un objeto y devuelve otro objeto diferente, normalmente derivado del parámetro.

```

1 [...]
2 Stream<Integer> edades = personas.stream().map(p -> p.getEdad());
3 //La función lambda hace que se añadan al stream de enteros las edades de las personas 'p' de la colección personas.

```

24.8.4.3. COMBINAR

Se pueden combinar operaciones intermedias (composición de funciones) para producir resultados más complejos. Por ejemplo, las edades de las personas adultas.

```

1 [...]
2 Stream<Integer> edadesAdultos = personas.stream()
3   .filter(p -> p.getEdad() >= 18).map (p -> p.getEdad());
4 //Añadiríamos al stream solamente las edades, de aquellas personas que son mayores de edad.

```

24.8.4.4. ORDENAR

El método `sorted` es una operación intermedia que permite ordenar los elementos de una colección según cierto criterio. Por ejemplo, ordenar las personas adultas por edad. `sorted` recibe como parámetro una interfaz `comparator`, que ya conocemos.

```

1 Stream<Persona> personasOrdenadas = personas.stream()
2   .filter(p -> p.getEdad() >= 18)
3   .sorted((p1, p2) -> p1.getEdad() - p2.getEdad());
4 //Para cada pareja de personas p1 y p2, ordénalas en función de la resta de la edad de p1 menos la edad de p2 (lo que hacíamos en el compareTo)

```

24.8.4.5. COLECCIÓN

El método `collect` es una operación final que permite obtener algún tipo de colección a partir de los datos procesados por las operaciones intermedias. Por ejemplo, una lista con las edades de las personas adultas.

```

1 List<Integer> edadesAdultos = personas.stream().filter(p -> p.getEdad() >= 18).map(p -> p.getEdad()).collect(Collectors.toList());
2 //similar a ejemplos anteriores, pero esta vez obtenemos una lista de enteros, en lugar de un stream.

```

El método `collect` también permite obtener una cadena de texto que une los elementos resultantes, a través de un separador común. En la función `Collectors.joining` se puede indicar también un prefijo y un sufijo para el texto.

```

1 String nombresAdultos = personas.stream().filter(p -> p.getEdad() >= 18)
2     .map(p -> p.getNombre())
3     .collect(Collectors.joining(", ", "Adultos: ", ""));
4 //genera una lista de nombres de personas, con un prefijo, separado y sufijo.

```

24.8.4.6. `forEach`

El método `forEach` permite recorrer cada elemento del stream resultante, y hacer lo que se necesite con él. Por ejemplo, sacar por pantalla en líneas separadas los nombres de las personas adultas.

```

1 personas.stream().filter(p -> p.getEdad() >= 18)
2     .map(p -> p.getNombre()).forEach(p -> System.out.println(p));

```

24.8.4.7. MEDIA ARITMÉTICA

El método `average` permite, junto con la operación intermedia `mapToInt`, obtener una media de un stream que haya producido una colección resultante numérica. Por ejemplo, la media de edades de las personas adultas.

```

1 double mediaAdultos = personas.stream().filter(p -> p.getEdad() >= 18)
2     .mapToInt(p -> p.getEdad()).average().getAsDouble();

```

24.9. Ejemplos UD07

24.9.1. Ejemplo01

```

1 package UD07.P2_2_Sets;
2
3 import java.util.TreeSet;
4
5 public class Ejemplo01 {
6
7     public static void main(String[] args) {
8         TreeSet<Integer> t = new TreeSet<>();
9         t.add(4);
10        t.add(3);
11        t.add(1);
12        t.add(99);
13        for (Integer i : t) {
14            System.out.print(i + " ");
15        }
16    }
17 }

```

24.9.2. Ejemplo02

```

1 package UD07.P2_2_Sets;
2
3 import java.util.LinkedHashSet;
4
5 public class Ejemplo02 {
6
7     public static void main(String[] args) {
8         LinkedHashSet<Integer> t = new LinkedHashSet<>();
9         t.add(4);
10        t.add(3);
11        t.add(1);
12        t.add(99);
13        for (Integer i : t) {
14            System.out.print(i + " ");
15        }
16    }
17 }

```

24.9.3. Ejemplo03

```

1 package UD07.P2_2_Sets;
2
3 import java.util.Collection;
4 import java.util.LinkedHashSet;
5 import java.util.TreeSet;
6
7 /**
8 *
9 * @author David Martínez (www.martinezpenya.es|ieseduardoprimo.es)
10 */
11 public class Ejemplo03 {
12
13     private static void imprimirColección(Collection<?> c) {
14         for (Object i : c) {
15             System.out.print(i.toString() + " ");
16         }
17         System.out.println("");
18     }
19
20     public static void main(String[] args) {
21         TreeSet<Integer> conjuntoA = new TreeSet<>();
22         conjuntoA.add(9);
23         conjuntoA.add(19);
24         conjuntoA.add(5);
25         conjuntoA.add(7); // Elementos del conjunto A: 9, 19, 5 y 7
26         LinkedHashSet<Integer> conjuntoB = new LinkedHashSet<>();
27         conjuntoB.add(10);
28         conjuntoB.add(20);
29         conjuntoB.add(5);
30         conjuntoB.add(7); // Elementos del conjunto B: 10, 20, 5 y 7
31
32         conjuntoA.addAll(conjuntoB);
33         imprimirColección(conjuntoA); //5 7 9 10 19 20
34
35         conjuntoA.removeAll(conjuntoB);
36         imprimirColección(conjuntoA); //9 19
37
38         //recolocamos todo como al principio
39         conjuntoA.add(5);
40         conjuntoA.add(7);
41         conjuntoB.add(10);
42         conjuntoB.add(20);
43         conjuntoB.add(5);
44         conjuntoB.add(7);
45
46         conjuntoA.retainAll(conjuntoB);
47         imprimirColección(conjuntoA); //5 7
48
49     }
50 }
```

24.9.4. Ejemplo04

Realiza un pequeño programa que pregunte al usuario 5 números diferentes (almacenándolos en un `HashSet`), y que después calcule la suma de los mismos (usando un bucle `for-each`).

Respuesta:

Una solución posible podría ser la siguiente. Fíjate en la solución y verás que el uso de conjuntos ha simplificado enormemente el ejercicio, permitiendo al programador o la programadora centrarse en otros aspectos:

```

1 package UD07.P2_HashSet;
2
3 import java.util.HashSet;
4 import java.util.Scanner;
5
6 public class EjemploHashSet {
7
8     public static void main(String[] args) {
9         HashSet<Integer> conjunto = new HashSet<Integer>();
10        Scanner teclado = new Scanner(System.in);
11        int numero;
12        do {
13            try {
14                System.out.print("Introduce un número " + (conjunto.size() + 1) + ": ");
15                numero = teclado.nextInt();
16                if (!conjunto.add(numero)) {
17                    System.out.println("Número ya en la lista. Debes introducir otro.");
18                }
19            } catch (NumberFormatException e) {
20                System.out.println("Número erróneo.");
21            }
22        } while (conjunto.size() < 5);
23        // Calcular la suma
24        Integer suma = 0;
25        for (Integer i : conjunto) {
26            suma = suma + i;
27        }
28        System.out.println("La suma es: " + suma);
29    }
30 }
```

24.9.5. Ejemplo05

```

1 package UD07.P2_3_Listas;
2
3 import java.util.ArrayList;
4 import java.util.Collection;
5 import java.util.LinkedList;
6
7 public class Ejemplo05 {
8
9     private static void imprimirColección(Collection<?> c) {
10        for (Object i : c) {
11            System.out.print(i.toString() + " ");
12        }
13        System.out.println("");
14    }
15    public static void main(String[] args) {
16        LinkedList<Integer> t = new LinkedList<>(); // Declaración y creación del LinkedList de enteros.
17        t.add(1); // Añade un elemento al final de la lista.
18        t.add(3); // Añade otro elemento al final de la lista.
19        t.add(1, 2); // Añade en la posición 1 el elemento 2.
20        t.add(t.get(1) + t.get(2)); // Suma los valores contenidos en la posición 1 y 2, y lo agrega al final.
21        t.remove(0); // Elimina el primer elementos de la lista.
22        imprimirColección(t); // 2 3 5
23
24        ArrayList<Integer> al = new ArrayList<>(); // Declaración y creación del ArrayList de enteros.
25        al.add(10);
26        al.add(11); // Añadimos dos elementos a la lista.
27        al.set(al.indexOf(11), 12); // Sustituimos el 11 por el 12, primero lo buscamos y luego lo reemplazamos.
28
29        al.addAll(0, t.subList(1, t.size()));
30        imprimirColección(al); // 3 5 10 12
31
32        al.subList(0, 2).clear();
33        imprimirColección(al); // 10 12
34    }
35 }
```

24.9.6. Ejemplo06

Tenemos la clase `Producto` con:

- Dos atributos: `nombre` (`String`) y `cantidad` (`int`).
- Un constructor con parámetros.
- Un constructor sin parámetros.
- Métodos `get` y `set` asociados a los atributos.

`Producto.java`

```
1 package UD07.P2_3_Listas;
2
3 public class Producto {
4
5     //Atributos
6     private String nombre;
7     private int cantidad;
8
9     //Métodos
10    //Constructor con parámetros donde asignamos el valor dado a los atributos
11    public Producto(String nombre, int cantidad) {
12        this.nombre = nombre;
13        this.cantidad = cantidad;
14    }
15
16    //Constructor sin parámetros donde inicializamos los atributos
17    public Producto() {
18        //La palabra reservada null se utiliza para inicializar los objetos,
19        //indicando que el puntero del objeto no apunta a ninguna dirección
20        //de memoria. No hay que olvidar que String es una clase.
21        this.nombre = null;
22        this.cantidad = 0;
23    }
24
25    //Método get y set
26    public String getNombre() {
27        return nombre;
28    }
29
30    public void setNombre(String nombre) {
31        this.nombre = nombre;
32    }
33
34    public int getCantidad() {
35        return cantidad;
36    }
37
38    public void setCantidad(int cantidad) {
39        this.cantidad = cantidad;
40    }
41 }
```

En el programa principal creamos una lista de productos y realizamos operaciones sobre ella:

Ejemplo06.java

```
1 package UD07.P2_3_Listas;
2
3 import java.util.ArrayList;
4
5 public class Ejemplo06 {
6
7     public static void main(String[] args) {
8
9         //Definimos 5 instancias de la clase Producto
10        Producto p1 = new Producto("Pan", 6);
11        Producto p2 = new Producto("Leche", 2);
12        Producto p3 = new Producto("Manzanas", 5);
13        Producto p4 = new Producto("Brocoli", 2);
14        Producto p5 = new Producto("Carne", 2);
15
16        //Definir un ArrayList
17        ArrayList<Producto> lista = new ArrayList<>();
18
19        //Colocar instancias de producto en ArrayList
20        lista.add(p1);
21        lista.add(p2);
22        lista.add(p3);
23        lista.add(p4);
24
25        //Añadimos "Carne" en la posición 1 de la lista
26        lista.add(1, p5);
27
28        //Añadimos "Carne" en la última posición
29        lista.add(p5);
30
31        //Imprimir el contenido del ArrayList
32        System.out.println(" - Lista con " + lista.size() + " elementos");
33
34        for (Producto p : lista) {
35            System.out.println(p.getNombre() + " : " + p.getCantidad());
36        }
37
38        p5.setCantidad(99); //cambiamos la cantidad al producto, cambiará la lista?
39
40        ((Producto)lista.get(1)).setCantidad(66); //
41
42        System.out.println(p5.getCantidad());
43
44        //Imprimir el contenido del ArrayList
45        System.out.println(" - Lista con " + lista.size() + " elementos");
46
47        for (Producto p : lista) {
48            System.out.println(p.getNombre() + " : " + p.getCantidad());
49        }
50
51        //Eliminar todos los valores del ArrayList
52        lista.clear();
53        System.out.println(" - Lista final con " + lista.size() + " elementos");
54    }
55 }
```

24.9.7. Ejemplo07

```

1 package UD07.P2_4_Maps;
2
3 import java.util.HashMap;
4
5 public class Ejemplo07 {
6
7     public static void main(String[] args) {
8         HashMap<String, Integer> hashMap = new HashMap<>();
9         //Insertamos un solo elemento A con valor 1
10        hashMap.put("A", 1);
11
12        //Busqueda por clave
13        if (hashMap.containsKey("A")) {
14            System.out.printf("Contiene la clave A. Su valor es: %d\n", hashMap.get("A"));
15        }
16
17        //Busqueda por valor
18        if (hashMap.containsValue(0)) {
19            System.out.println("Contiene el valor 0");
20        }
21
22        //Eliminar el elemento con clave A
23        hashMap.remove("A");
24
25        //Ahora añadimos varios elementos para imprimirlos
26        hashMap.put("A", 1);
27        hashMap.put("E", 12);
28        hashMap.put("I", 15);
29        hashMap.put("O", 0);
30        hashMap.put("U", 0);
31        //Recorremos el mapa y lo imprimimos
32        for (HashMap.Entry<String, Integer> entry : hashMap.entrySet()) {
33            System.out.printf("Clave: %. Valor: %d\n", entry.getKey(), entry.getValue());
34        }
35    }
36 }
```

24.9.8. Ejemplo08

Ejemplo que crea, rellena y recorre un `ArrayList` de dos formas diferentes. Cabe destacar que, por defecto, el método `System.out.println()` invoca al método `toString()` de los elementos que se le pasen como argumento, por lo que realmente no es necesario utilizar `toString()` dentro de `println()`.

```
1 package UD07.P3.Iterators;
2
3 import java.util.ArrayList;
4 import java.util.Iterator;
5
6 public class Ejemplo08 {
7
8     public static void main(String[] args) {
9         //Creamos la lista
10        ArrayList l = new ArrayList();
11
12        //Añadimos elementos al final de la lista
13        l.add("uno");
14        l.add("dos");
15        l.add("tres");
16        l.add("cuatro");
17
18        //Añadimos el elemento en la posición 2
19        l.add(2, "dos2");
20
21        System.out.println(l.size()); //devuelve 5
22        System.out.println(l.get(0)); //devuelve uno
23        System.out.println(l.get(1)); //devuelve dos
24        System.out.println(l.get(2)); //devuelve dos2
25        System.out.println(l.get(3)); //devuelve tres
26        System.out.println(l.get(4)); //devuelve cuatro
27
28        //Recorremos la lista con un for y mostramos su contenido
29        for (int i = 0; i < l.size(); i++) {
30            System.out.print(l.get(i));
31        } //Imprime: unodosdos2trescuatro
32
33        System.out.println();
34
35        //Recorremos la lista con un Iterador
36        //Creamos el iterador
37        Iterator it = l.iterator();
38
39        //mientras haya elementos
40        while (it.hasNext()) {
41            System.out.print(it.next()); //obtengo el siguiente elemento
42        } //Imprime: unodosdos2trescuatro
43
44        System.out.println();
45
46        for (Object s : l) {
47            System.out.print(s);
48        } //Imprime: unodosdos2trescuatro
49    }
50 }
```

24.9.9. Ejemplo09

```

1 package UD07.P3.Iterators;
2
3 import UD07.P2_3_Listas.Producto;
4 import java.util.ArrayList;
5 import java.util.Iterator;
6
7 public class Ejemplo09 {
8
9     public static void main(String[] args) {
10
11         //Definimos 5 instancias de la clase Producto
12         Producto p1 = new Producto("Pan", 6);
13         Producto p2 = new Producto("Leche", 2);
14         Producto p3 = new Producto("Manzanas", 5);
15         Producto p4 = new Producto("Brocoli", 2);
16         Producto p5 = new Producto("Carne", 2);
17
18         //Definir un ArrayList
19         ArrayList<Producto> lista = new ArrayList<>();
20
21         //Colocar instancias de producto en ArrayList
22         lista.add(p1);
23         lista.add(p2);
24         lista.add(p3);
25         lista.add(p4);
26
27         //Añadimos "Carne" en la posición 1 de la lista
28         lista.add(1, p5);
29
30         //Añadimos "Carne" en la última posición
31         lista.add(p5);
32
33         //Imprimir el contenido del ArrayList
34         System.out.println(" - Lista con " + lista.size() + " elementos");
35
36         //Definir Iterator para extraer/imprimir valores
37         //si queremos utilizar un for con el iterador no hace falta poner el incremento
38         for (Iterator<Producto> it = lista.iterator(); it.hasNext(); ) {
39             Producto p = it.next();
40             System.out.println(p.getNombre() + " : " + p.getCantidad());
41         }
42
43         p5.setCantidad(99); //cambiamos la cantidad al producto, cambiará la lista?
44
45         ((Producto)lista.get(1)).setCantidad(66); //
46
47         System.out.println(p5.getCantidad());
48
49         //Imprimir el contenido del ArrayList
50         System.out.println(" - Lista con " + lista.size() + " elementos");
51
52         //Definir Iterator para extraer/imprimir valores
53         //si queremos utilizar un for con el iterador no hace falta poner el incremento
54         for (Iterator<Producto> it = lista.iterator(); it.hasNext(); ) {
55             Producto p = it.next();
56             System.out.println(p.getNombre() + " : " + p.getCantidad());
57         }
58
59         //Eliminar todos los valores del ArrayList
60         lista.clear();
61         System.out.println(" - Lista final con " + lista.size() + " elementos");
62     }
63 }
```

24.9.10. Ejemplo10

Ejercicio resuelto `Comparator1`. Imagínate que Objeto es una clase como la siguiente:

```

1 package UD07.P4.Comparator1;
2
3 public class Objeto {
4
5     public int a;
6     public int b;
7
8     public Objeto(int a, int b) {
9         this.a = a;
10        this.b = b;
11    }
12
13    @Override
14    public String toString() {
15        return "Objeto{" + "a=" + a + ", b=" + b + '}';
16    }
17
18 }
```

Imagina que ahora, al añadirlos en un `TreeSet`, estos se tienen que ordenar de forma que la suma de sus atributos (**a** y **b**) sea descendente, ¿como sería el comparador?

Respuesta

Una de las posibles soluciones a este problema podría ser la siguiente:

```

1 package UD07.P4.Comparador1;
2
3 import java.util.Comparator;
4
5 class ComparadorDeObjetos implements Comparator<Objeto> {
6
7     @Override
8     public int compare(Objeto o1, Objeto o2) {
9         int sumao1 = o1.a + o1.b;
10        int sumao2 = o2.a + o2.b;
11        if (sumao1 < sumao2) {
12            return -1;
13        } else if (sumao1 > sumao2) {
14            return 1;
15        } else {
16            return 0;
17        }
18    }
19 }
```

Y para usarlo tendriamos:

```

1 package UD07.P4.Comparador1;
2
3 import java.util.TreeSet;
4
5 public class Principal {
6
7     public static void main(String[] args) {
8         TreeSet<Objeto> ts = new TreeSet<Objeto>(new ComparadorDeObjetos());
9
10        Objeto o1= new Objeto(0, 1);
11        ts.add(o1);
12
13        ts.add(new Objeto(1, 2));
14        ts.add(new Objeto(4, 5));
15        ts.add(new Objeto(2, 3));
16
17        for (Objeto o : ts) {
18            System.out.println(o);
19        }
20    }
21 }
```

Observa que la salida muestra los elementos correctamente ordenados, aunque se insertaron de manera "aleatoria":

```

1 Objeto{a=4, b=5}
2 Objeto{a=2, b=3}
3 Objeto{a=1, b=2}
4 Objeto{a=0, b=1}
```

24.9.11. Ejemplo11

Ejercicio resuelto `Comparator2`. Ahora convertiremos la clase `Objeto` para que directamente implemente la interfaz `Comparable`:

```

1 package UD07.P4.Comparator2;
2
3 public class Objeto implements Comparable<Objeto> {
4
5     public int a;
6     public int b;
7
8     public Objeto(int a, int b) {
9         this.a = a;
10        this.b = b;
11    }
12
13    @Override
14    public String toString() {
15        return "Objeto{" + "a=" + a + ", b=" + b + '}';
16    }
17
18    @Override
19    public int compareTo(Objeto t) {
20        int sumao1 = this.a + this.b;
21        int sumao2 = t.a + t.b;
22        if (sumao1 < sumao2) {
23            return -1;
24        } else if (sumao1 > sumao2) {
25            return 1;
26        } else {
27            return 0;
28        }
29    }
30}
31

```

Y lo usamos directamente en la clase Principal:

```

1 package UD07.P4.Comparator2;
2
3 import java.util.TreeSet;
4
5 public class Principal {
6
7     public static void main(String[] args) {
8         TreeSet<Objeto> ts = new TreeSet<Objeto>();
9
10        ts.add(new Objeto(0, 1));
11        ts.add(new Objeto(1, 2));
12        ts.add(new Objeto(4, 5));
13        ts.add(new Objeto(2, 3));
14
15        for (Objeto o : ts) {
16            System.out.println(o);
17        }
18    }
19}

```

Fíjate que la salida sigue mostrando los elementos correctamente ordenados, aunque se insertaron de manera "aleatoria":

```

1 Objeto{a=4, b=5}
2 Objeto{a=2, b=3}
3 Objeto{a=1, b=2}
4 Objeto{a=0, b=1}

```

24.9.12. Ejemplo 12

```

1 package UD07.P6_Funcional;
2
3 import java.util.ArrayList;
4 import java.util.List;
5 import java.util.stream.Collectors;
6
7 class Persona{
8
9     private String nombre;
10    private int edad;
11
12    public Persona(String nombre, int edad) {
13        this.nombre = nombre;
14        this.edad = edad;
15    }
16
17    public String getNombre() {
18        return nombre;
19    }
20
21    public int getEdad() {
22        return edad;
23    }
24
25    public String toString(){
26        return nombre + " " + edad;
27    }
28}
29
30 public class P6_2_ImperativosFuncional {
31     public static void main(String[] args) {
32
33         List<Persona> personas = new ArrayList<>();
34
35         personas.add(new Persona("Nacho", 52));
36         personas.add(new Persona("David", 47));
37         personas.add(new Persona("Pepe", 42));
38         personas.add(new Persona("Maria", 22));
39         personas.add(new Persona("Marta", 4));
40
41         //IMPERATIVO
42         List<Persona> adultosImperativo = new ArrayList<>();
43         for (int i = 0; i < personas.size(); i++) {
44             if (personas.get(i).getEdad() >= 18)
45                 adultosImperativo.add(personas.get(i));
46         }
47         for (Persona p : adultosImperativo) {
48             System.out.println(p.getNombre() + " " + p.getEdad());
49         }
50
51         //FUNCIONAL
52         List<Persona> adultosDeclarativo = personas.stream().filter(p -> p.getEdad() >= 18).collect(Collectors.toList());
53         for (Persona p : adultosDeclarativo) {
54             System.out.println(p.getNombre() + " " + p.getEdad());
55         }
56     }
57 }
```

24.9.13. Ejemplo 13

```
1 package UD07.P6_Funcional;
2
3 import java.util.ArrayList;
4 import java.util.Comparator;
5 import java.util.List;
6 class ComparadorPersona implements Comparator<Persona> {
7     @Override
8     public int compare(Persona p1, Persona p2){
9         return p2.getEdad() - p1.getEdad();
10    }
11 }
12 public class P6_3_Lambda {
13     public static void main(String[] args) {
14
15         List<Persona> personas = new ArrayList<>();
16
17         personas.add(new Persona("Nacho", 52));
18         personas.add(new Persona("David", 47));
19         personas.add(new Persona("Pepe", 42));
20         personas.add(new Persona("Maria", 22));
21         personas.add(new Persona("Marta", 4));
22
23         //IMPERATIVO
24         personas.sort(new ComparadorPersona());
25         for (int i = 0; i < personas.size(); i++){
26             System.out.println(personas.get(i).toString());
27         }
28
29         //FUNCIONAL
30         personas.sort((p1, p2) -> p2.getEdad() - p1.getEdad());
31         for (int i = 0; i < personas.size(); i++){
32             System.out.println(personas.get(i).toString());
33         }
34     }
35 }
```

24.9.14. Ejemplo 14

```

1 package UD07.P6_Funcional;
2
3 import java.util.ArrayList;
4 import java.util.List;
5 import java.util.stream.Collectors;
6 import java.util.stream.Stream;
7
8 public class P6_4_ColeccionesStreams {
9
10    public static void main(String[] args) {
11        ArrayList<Persona> personas = new ArrayList<>();
12        personas.add(new Persona("Nacho", 52));
13        personas.add(new Persona("David", 47));
14        personas.add(new Persona("Pepe", 42));
15        personas.add(new Persona("Maria", 22));
16        personas.add(new Persona("Marta", 4));
17
18        // FILTRADO
19        System.out.println("FILTRADO");
20        Stream<Persona> adultos = personas.stream().filter(p -> p.getEdad() >= 18);
21        //La función lambda se podría traducir como: "Aquellas personas 'p' de la
22        // colección cuya edad sea mayor o igual que 18 años.
23        adultos.forEach(p -> System.out.println(p.getNombre() + " " + p.getEdad()));
24
25        // MAPEO
26        System.out.println("MAPEO");
27        Stream<Integer> edades = personas.stream().map(p -> p.getEdad());
28        //La función lambda hace que se añadan al stream de enteros las edades de las
29        // personas 'p' de la colección personas.
30        edades.forEach(e -> System.out.println(e));
31
32        // COMBINAR
33        System.out.println("COMBINAR");
34        Stream<Integer> edadesAdultos = personas.stream()
35            .filter(p -> p.getEdad() >= 18).map(p -> p.getEdad());
36        //Añadiríamos al stream solamente las edades, de aquellas personas que son mayores de edad.
37        edadesAdultos.forEach(e -> System.out.println(e));
38
39        // ORDENAR
40        System.out.println("ORDENAR");
41        Stream<Persona> personasOrdenadas = personas.stream()
42            .filter(p -> p.getEdad() >= 18)
43            .sorted((p1, p2) -> p1.getEdad() - p2.getEdad());
44        //Para cada pareja de personas p1 y p2, ordénalas en función de la resta
45        // de la edad de p1 menos la edad de p2 (lo que hacíamos en el compareTo)
46        personasOrdenadas.forEach(p -> System.out.println(p.getNombre() + " " + p.getEdad()));
47
48        // COLECCIÓN
49        System.out.println("COLECCIÓN");
50        List<Integer> edadesAdultos2 = personas.stream()
51            .filter(p -> p.getEdad() >= 18)
52            .map(p -> p.getEdad()).collect(Collectors.toList());
53        //similar a ejemplos anteriores, pero esta vez obtenemos una lista de enteros,
54        // en lugar de un stream.
55        System.out.println("Lista: " + edadesAdultos2);
56
56        String nombresAdultos = personas.stream().filter(p -> p.getEdad() >= 18)
57            .map(p -> p.getNombre())
58            .collect(Collectors.joining(", ", "Adultos: ", ""));
59        //genera una lista de nombres de personas, con un prefijo, separado y sufijo.
60        System.out.println("String: " + nombresAdultos);
61
62        // FOREACH
63        System.out.println("FOREACH");
64        personas.stream().filter(p -> p.getEdad() >= 18)
65            .map(p -> p.getNombre()).forEach(p -> System.out.println(p));
66
67        // MEDIA
68        System.out.println("MEDIA");
69        double mediaAdultos = personas.stream().filter(p -> p.getEdad() >= 18)
70            .mapToInt(p -> p.getEdad()).average().getAsDouble();
71        System.out.println("Media adultos: " + mediaAdultos);
72    }
73 }
74 }
```

24.10. Píldoras informáticas relacionadas

⌚19 de febrero de 2026

25. 8.2 Ejercicios de la UD07

25.1. Ejercicios

1. (`package Varios`) Diseñar la clase `Varios` con los siguientes métodos **estáticos** que se harán apoyándose en alguna clase de las vistas al estudiar las colecciones de Java:
 - `int[] quitarDuplicados (int[] v)`, que dado un array de enteros devuelva otro array con los mismos valores que el original pero sin duplicados.
 - `int[] union1(int[] v1, int[] v2)`, que dados dos arrays `v1` y `v2` devuelva otro array con los elementos que están en `v1` o que están en `v2`, sin que ningún elemento se repita.
 - `int[] union2(int v1[], int v2[])`, que dados dos arrays `v1` y `v2` devuelva otro array con los elementos que están en `v1` o que están en `v2`. En este caso, si hay elementos duplicados se mantendrán.
 - `int[] interseccion(int v1[], int v2[])`, que dados dos arrays `v1` y `v2` devuelva otro array con los elementos que aparecen en los dos arrays. Cada elemento común aparecerá una sola vez en el resultado.
 - `int[] diferencia1 (int v1[], int v2[])`, que dados dos arrays `v1` y `v2` devuelva otro array con los elementos de `v1` que no están en `v2`. En caso de haber elementos duplicados en `v1` estos se mantendrán en el resultado.
 - `int[] diferencia2 (int v1[], int v2[])`, que dados dos arrays `v1` y `v2` devuelva otro array con los elementos de `v1` que no están en `v2`. El array resultante no tendrá elementos duplicados.
2. (`package Biblioteca`) Se quiere hacer una aplicación en la que los usuarios van a hacer búsquedas de libros para saber si se encuentran en los fondos de la biblioteca. El funcionamiento básico sería algo así: Al iniciarse la aplicación todo el catálogo de libros se cargaría en memoria y a partir de ese momento los usuarios pueden realizar búsquedas por título, que interesa que sean lo más rápidas posibles. Nunca se insertan nuevos libros durante la ejecución de la aplicación.
 - Diseña la clase `Libro` con los métodos que consideres oportunos y los siguientes atributos:
 - `Titulo (String)`: Es el dato que identifica al libro.
 - `Autor (String)`: Autor del libro.
 - `Estanteria (String)`: Estantería de la biblioteca en la que se encuentra el libro.
 - Diseña la clase `CatalogoLibros` como una colección de libros (llamada `catalogo`). Utiliza el tipo de colección que crees que más se ajusta a los requisitos de la aplicación justificando la elección. Implementa los siguientes métodos:
 - `public CatalogoLibros(Libro v[])`: Constructor. Para simplificar, inicializa el catálogo y lo rellena con los libros del `array v`, en lugar de obtenerlos de un fichero.
 - `public String buscar(Libro l)`: Dado un libro, lo busca en el Catálogo y devuelve la estantería en la que se encuentra el libro o `null` si el libro no está en el Catálogo.
3. (`package Academia`) Se quiere diseñar una clase `Academia`. De una Academia se conoce su nombre, dirección y las Aulas que tiene (Necesitas también generar la clase `Aula`).

Definir la clase `Academia` utilizando una **Collection** para almacenar las aulas. El tipo de colección a utilizar se decidirá teniendo en cuenta que éstas se quieren mantener ordenadas según el criterio del método `compareTo` de la clase `Aula`. Implementar los atributos, el constructor, y los siguientes métodos:

 - `void ampliar (Aula a)`, que añade un aula a la academia.
 - `void quitar (Aula a)`, que elimina un aula de la academia.
 - `int getNumAulas()`, que devuelva el número de aulas que tiene.
 - `toString()`, que muestre todas las aulas de la academia.
4. (`paquete ListaEspera`) Deseamos mantener la lista de espera de pacientes de un hospital.

Se quiere poder:

 - Añadir pacientes a la lista.
 - Mantener los pacientes por orden de inserción en la lista de espera.
 - Obtener el paciente más prioritario (al que hay que atender de la lista de espera), es decir, el que más tiempo lleva esperando, el que antes entró en el hospital.

Pasos a seguir:

 - a. Diseña la clase `Paciente` con los atributos nombre y gravedad. La gravedad es un valor aleatorio (entre 1: más grave y 5: menos grave) generado al crear el paciente.
 - b. Realiza la implementación de la clase `ListaEspera` con la estructura de datos elegida. Añade los métodos para insertar un paciente, obtener de la lista de espera al más prioritario y eliminar de la lista de espera al más prioritario. ¿Qué tipo de

estructura de datos utilizaríamos para almacenar los pacientes, un `List`, `Set` o `Map`? Justifica la respuesta (puntos a favor de la que elijes y en contra de las que descartas).

- c. Implementa una `ListaEsperaPorGravedad`, en la que se atienda primero a los pacientes más graves, independientemente de si llegaron antes o después.

5. (`paquete DiccionarioIngEsp`)

Diseñar la clase `DiccionarioBilingüe` para almacenar pares formados por:

- a. Palabra en castellano.

- b. Colección de traducciones a inglés.

La clase dispondrá de los siguientes métodos:

- Constructor: crea el diccionario vacío.
- `anyadirTraducción(String cast, String ingl)`: Añade la pareja (cast, ingl) al diccionario de forma que:
 - Si la palabra `cast` no estaba en el diccionario la añade, junto con su traducción.
 - Si la palabra `cast` estaba ya en el diccionario pero no aparecía como traducción la palabra `ingl`, añade `ingl` a su colección de traducciones.
 - Si la palabra `cast` estaba y la traducción `ingl` también, no se realizarán cambios.
 - El método devuelve `true` si se han realizado cambios en el diccionario y `false` en caso contrario.
- `QuitarTraducción(String cast, String ingl)`: Quita la traducción `ingl` a la palabra `cast`. Si la palabra en castellano se queda sin traducciones, se elimina del diccionario. Si se han producido cambios se devuelve `true` y en caso contrario `false`.
- `traduccionesDe(String cast)`: Devuelve una colección con las traducciones de la palabra indicada o `null` si la palabra no está en el diccionario.
- `toString()`: Devuelve un `String` con las palabras del diccionario y sus traducciones.
- (clase `RepeticiónPalabras`) Escribe un programa que abra el fichero de texto que indique el usuario y muestre cuántas veces se repite cada palabra que contiene. Ayudarse de un `Map`. ¿Se podría resolver con un `Set`? ¿Y con un `List`?
- (clase `TraductorSimple`) Escribir un programa que solicite al usuario una frase y la muestre traducida a inglés, palabra a palabra. Para ello, se dispone de un fichero `palabras.txt` que contiene parejas (palabra en español, palabra en inglés) separadas por un tabulador. Cada pareja se encuentra en una línea del fichero. El proceso será el siguiente:
 - Leer el fichero y cargar sus datos en una estructura de datos adecuada. Tener en cuenta que nos interesará buscar una palabra en castellano y obtener su correspondencia en inglés.
 - Solicitar al usuario una frase. Traducir, usando la estructura de datos anterior, cada palabra de la frase y formar con ellas la frase traducida.
 - Mostrar la frase traducida al usuario.

6. (`paquete ListaAdmitidos`) Una serie de personas han solicitado realizar un curso de inglés. De las que han sido admitidas se quiere almacenar su nif, su nombre y su nivel en un `HashSet`. En el `HashSet` se almacenarán objetos de la clase `Inscripción`

- Implementa la clase `Inscripción` para representar el nif, nombre y nivel de un solicitante. Además de los atributos implementa aquellos métodos que consideres necesarios.
- Escribe un programa (clase `comprobarAdmisión`) que
 - Defina un `HashSet` de `Inscripciones`, llamado `admitidas`.
 - Añada varias inscripciones (invéntate los datos).
 - Permita al usuario introducir un dni para comprobar si la persona indicada ha sido admitida. Indicarle si aparece o no en la lista y, en caso afirmativo, mostrar el nombre y el nivel en que ha sido admitido.

7. (`clase PalabrasOrdenadas`) Escribe un programa que, dado un fichero de texto cuya ubicación indica el usuario, muestre sus palabras ordenadas ascendente y, después, descendente. Cada palabra se mostrará una sola vez, aunque en el texto aparezca varias.

25.2. Actividades

1. **Actividad 1.** Realizar las siguientes actividades relacionadas con `ArrayList`.

- Crear un `ArrayList` de enteros llamado `misNumeros`.
- Añadir los valores 1, 6, 3, 2, 0, 4, 5.
- Mostrar los datos del `ArrayList`.

- Mostrar el valor de la posición 5.
- Añadir el valor 8 en la posición 4.
- Cambiar el valor de la posición 1 por 9.
- Eliminar el valor 5. (`misNumeros.remove(new Integer(5))`)
- Eliminar el valor de la posición 3.
- Recorrer el array con un bucle `for`.
- Recorrer el array con un bucle `Iterator`.
- Comprobar si existe el elemento 0.
- Comprobar si existe el elemento 7.
- Clonar el `ArrayList` `misNumeros` en otro llamado `copiaArrayList`.
- Añadir el elemento 9.
- Mostrar la posición de la primera ocurrencia del elemento 9.
- Mostrar la posición de la última ocurrencia del elemento 9.
- Borrar todos los elementos del `ArrayList` `copiaArrayList`.
- Comprobar si el `ArrayList` `copiaArrayList` está vacío.
- Convertir el `ArrayList` `misNumeros` en un `Array` y recorrerlo con un bucle mejorado.

2. Actividad 2. Un cine precisa una aplicación para controlar las personas de la cola para los estrenos de películas. Debemos crear una lista con la edad de las personas de la cola y tendremos que calcular la entrada según la edad de la persona (mínimo 5 años). Para la edad de la persona se generan aleatoriamente números entre 5 y 60 años. Al final, deberemos mostrar la cantidad total recaudada. El número de personas de la cola se elige al azar entre 0 y 50.

La lista de precios se basa en la siguiente tabla.

EDAD	PRECIO
Entre 5 y 10 años	5 €
Entre 11 y 17 años	7.5 €
Mayor de 18 años	9.5 €

Como comprobación imprime el número de personas, el precio total y la lista de edades. Por ejemplo:

```

1 Hay un total de 6 personas en la cola.
2 El precio total es de 57,00 euros
3 [18, 36, 50, 35, 28, 55]

```

3. Actividad 3. Un supermercado nos pide que hagamos una aplicación que almacene los productos comprados. La aplicación debe almacenar `Productos` (clase `Actividad3Producto`) y cada producto al crearse contiene una `cantidad`, un `precio` (generados aleatoriamente). El nombre del producto será básico (`producto1`, `producto2`, `producto3`, etc.). Calcular el precio total de una lista de entre 1 y 10 productos (aleatorio). Mostrar un ticket con todo lo vendido y el precio final.

```

1 Producto1 3 47,95 143,84
2 Producto2 7 84,37 590,62
3 Producto3 7 33,43 234,04
4 Producto4 2 95,42 190,84
5 Producto5 10 53,50 534,96
6 Producto6 3 26,21 78,62
7 El precio final del ticket es de 1772,92 euros

```

4. Actividad 4. Desarrollar un sistema de gestión de pacientes. Tendremos un archivador dónde iremos guardando todas las fichas de los pacientes. Las fichas (`Actividad4Paciente`) contienen la siguiente información: `nombre`, `apellidos` y `edad`. La clase `Actividad4` tiene un método `main` en el que se crea un archivador, dos o tres fichas que se guardarán en el archivador, se listará el contenido, se recorrerá el archivador borrando todas las fichas que cumplan dos condiciones (nombre "David" o menor de 10 años, por ejemplo) y por último se eliminará alguna ficha (directamente) y se volverá a listar su contenido.

```

1 LISTADO DE PACIENTES
2 =====
3 {nombre=David, apellidos=Martinez Peña, edad=35}
4 {nombre=Elena, apellidos=Garcia Perez, edad=15}
5 {nombre=Marc, apellidos=Redondo Perez, edad=5}
6 {nombre=Pepe, apellidos=Perez Peña, edad=16}
7
8 LISTADO DE PACIENTES
9 =====
10 {nombre=Elena, apellidos=Garcia Perez, edad=15}
11 {nombre=Pepe, apellidos=Perez Peña, edad=16}
12
13 LISTADO DE PACIENTES
14 =====
15 {nombre=Elena, apellidos=Garcia Perez, edad=15}

```

5. Actividad 5. Crear una estructura Map llamada divisas, que almacene pares de moneda y valor al cambio en euros. Por ejemplo Dólar: 0,81€

- Añadir los siguientes pares moneda/valor al Map divisas:

Moneda	Valor en €
Dólar Americano	0.81
Franco Suizo	0.85
Libra Esterlina	1.14
Corona Danesa	0.13
Peso Mexicano	0.04
Dólar Singapur	0.62
Real Brasil	0.24

- Mostrar el valor de la Libra Esterlina.
- Mostrar todas las divisas con las que se opera y su valor.
- Indicar el número de divisas del Map.
- Eliminar la divisa Real Brasil y mostrar los datos del Map.
- Mostrar si existe la divisa Peso Mexicano.
- Mostrar si existe la divisa Euro.
- Mostrar si existe el valor al cambio 0.85 €.
- Mostrar si existe el valor al cambio 0.33 €.
- Indicar si el Map divisas está vacío.
- Borra todos los componentes del Map divisas.
- Volver a indicar si el Map divisas está vacío.

```

1 1.14
2 Franco Suizo --> 0,85
3 Corona Danesa --> 0,13
4 Peso Mexicano --> 0,04
5 Dólar Americano --> 0,81
6 Libra Esterlina --> 1,14
7 Real Brasil --> 0,24
8 Dólar Singapur --> 0,62
9
10 Franco Suizo --> 0,85
11 Corona Danesa --> 0,13
12 Peso Mexicano --> 0,04
13 Dólar Americano --> 0,81
14 Libra Esterlina --> 1,14
15 Dólar Singapur --> 0,62
16 SI existe Peso Mexicano
17 NO existe Euro €
18 SI existe el valor 0,85
19 NO existe el valor 0,33
20 NO está vacío
21 Está vacío

```

25.3. Ejercicios Genericidad

1. Crear una clase Genericos (proyecto Genéricos, paquete Genericos) que incorpore los métodos genéricos que se indican a continuación. Los métodos creados serán *public static*. En el proyecto se creará además la clase o clases necesarias para probar los métodos desarrollados.
 - a. `Object minimo (Object o1, Object o2)`, que devuelva el mínimo de dos objetos cualesquiera (que se suponen del mismo tipo). Una vez desarrollado, prueba el método para obtener el mínimo de dos objetos Integer. Pruebalo también para obtener el mínimo entre un Objeto Integer y un objeto String. En éste último caso, el programa ¿da error de ejecución? Si es así, explica por qué.
 - b. `Object maximo (Object o1, Object o2)`, que devuelva el maximo de dos objetos cualesquiera (que se suponen del mismo tipo).
 - c. `Object minimo (Object v[])`, que devuelva el mínimo de un array de objetos cualesquiera (que se suponen del mismo tipo). Al respecto de éste último comentario, ¿Se puede poner en un array de Object objetos de distinto tipo, como por ejemplo Strings, Integer, ...? En caso afirmativo, ¿funcionaría el método desarrollado con un array construido así?
 - d. `Object maximo (Object v[])`, que devuelva el maximo de un array de objetos cualesquiera (que se suponen del mismo tipo).
 - e. `int numVeces(Object v[], Object x)` que devuelva el el numero de apariciones del objeto x en el array v.
 - f. `int numVecesOrdenado(Object v[], Object x)` que devuelva el el numero de apariciones del objeto x en el array v **ordenado ascendentemente**.
 - g. `int mayores(Object v[], Object x)` que, dado un array de Object v y un Object x devuelva el número de elementos de v que son mayores que x.
 - h. `int mayoresOrdenado(Object v[], Object x)` que, dado un array de Object v **ordenado ascendentemente** y un Object x devuelva el número de elementos de v que son mayores que x.
 - i. `int menores(Object v[], Object x)` que, dado un array de Object v y un Object x devuelva el número de elementos de v que son menores que x.
 - j. `int menoresOrdenado(Object v[], Object x)` que, dado un array de Object v **ordenado ascendentemente** y un Object x devuelva el número de elementos de v que son menores que x.
 - k. `boolean estaEn(Object v[], Object x)` que devuelva true si el Objeto x está en el array v.
 - l. `boolean estaEnOrdenado(Object v[], Object x)` que devuelva true si el Objeto x está en el array v, **ordenado ascendentemente**.
 - m. `int posiciónDe(Object v[], Object x)`, que devuelva la posición que ocupa x dentro del array v, o -1 si x no está en v.
 - n. `int posicionDeOrdenado(Object v[], Object x)`, que devuelva la posición que ocupa x dentro del array v **ordenado ascendentemente**, o -1 si x no está en v.
 - o. `boolean estaOrdenado(Object v[])`, que devuelva true si el array está ordenado ascendentemente.
2. (**paquete Nevera**) Se quiere crear una aplicación que controla una nevera inteligente de última generación. Los alimentos que contiene la nevera se van a representar como objetos de la clase `Alimento` y la clase `NeveraInteligente`, tiene un array de Alimentos entre sus atributos privados.

Se pide implementar la clase `Alimento` teniendo en cuenta que uno de los métodos de `NeveraInteligente` necesitará ordenar **por calorías** los Alimentos que contiene la nevera utilizando un método de Ordenación genérico. El diseño de la clase `Alimento` ha de incluir, por tanto, determinados elementos que lo permitan. La clase `Alimento` tendrá únicamente dos atributos (privados): nombre y calorías.
3. (**paquete Academia**) Se quiere diseñar una clase `Academia`. De una Academia se conoce su nombre, dirección y las `Aulas` que tiene (`Aula` es una clase implementada en un ejercicio de herencia que ya hicimos).
 - a. Definir la clase `Academia` utilizando una colección (que permita ordenación) para almacenar las aulas.: Implementar los atributos, el constructor, y los siguientes métodos: - `void ampliar (Aula a)`, que añade un aula a la academia. - `void quitar (Aula a)`, que elimina un aula de la academia. - `int getNumAulas()`, que devuelva el número de aulas que tiene. - método `toString()`
 - b. Realiza en la clase `Aula` los cambios necesarios para que se pueda ordenar las aulas de la Academia usando un método genérico de ordenación. El orden sería creciente por capacidad del aula., y a igual capacidad primero las aulas de mayor superficie
 - c. Añade a la clase `Academia` un método `ordenar` que ordene las aulas con el criterio especificado. Para realizar la ordenación se llamará a un método de ordenación.
4. (**paquete Conjuntos**)
 - a. Diseñar un **interface Conjunto** para modelizar conjuntos de elementos. Diseñar (solo la cabecera) de los siguientes métodos de la clase conjunto (prestar atención a si los métodos deben ser *static* o no):
 - `Object add (Object e)`
 - `Object remove (Object e)`
 - `Object contains (Object e)`
 - `Object size ()`
 - `Object clear ()`
 - `Object clone ()`

- `Añadir`, que añade un elemento al conjunto, provocando la excepción `ElementoDuplicado` si el elemento ya estaba en el conjunto.
 - `Quitar`, que elimina el elemento indicado al conjunto. Provoca `ElementoNoEncontrado` si el elemento indicado no estaba en el conjunto.
 - `Intersección`, que dados dos conjuntos que recibe como parámetro devuelve un tercer conjunto que es la intersección de los dos datos.
 - `Pertenece`, que dado un elemento devuelve si este pertenece o no al conjunto.
- b. Diseñar una clase `ConjuntoArray` que implemente el interface `Conjunto`. Esta clase implementará los métodos del interface `Conjunto`. Para ello utilizará un array `Object elementos[]` y un `int numElementos`, de manera que los elementos del conjunto se mantendrán almacenados en el array. Además de los métodos del interface habrá que crear un constructor para la clase y también vendrá bien tener un método `toString` para poder probarla.

NOTA: También lo puedes implementar con una Colección de las vistas en el tema anterior.

25.4. Programación funcional. Funciones Lambda.

1. Supongamos que tenemos implementada una clase `Libro` que tiene los atributos de `titulo`, `autor` y `precio`, con el correspondiente constructor y sus getters. Disponemos además de una lista de libros en una variable `libros`. Se pide implementar un método que muestre los títulos de los libros por pantalla, ordenados de menor a mayor.
 2. Disponemos de una clase llamada `Receta` que almacena los datos de una receta de cocina: su nombre, su categoría (carnes, pastas...) y sus calorías, incluyendo su constructor y getters. Supongamos que tenemos una lista de recetas llamada `recetas`. Se pide implementar las siguientes consultas en Java:
 - Recetas de menos de 500 calorías
 - Nombres de las recetas de "carnes", ordenadas alfabéticamente
 - Media de calorías de las recetas de "verduras"
 - Cuántas recetas hay de más de 800 calorías
 3. Usando los streams de Java, que están muy relacionados con las expresiones lambda, debes crear una lista de 1000 números enteros aleatorios entre -5000 y 5000 y a partir de dicha lista vamos a imprimir por pantalla:
 - El máximo de los números pares.
 - El mínimo de los números múltiplos de 3.
 - El total de números negativos.
 - El total de números primos.
 - El máximo número primo.
 4. Paquete `GranjaDAM`, crea una clase `Animal` con los siguientes atributos, constructor con todos los atributos, getters y setters:
 - String especie
 - String genero
 - String raza
 - String alimentacion
 - float peso
 - String color
 - boolean tienePelo

Crea una clase `Granja` la cual contendrá un `List<Animal>` con los animales que posee. Crea también los métodos siguientes:

 - `void nuevoAnimal (Animal a)`
 - `int contarAnimalesDeRaza(String raza)`
 - `Map<String, Integer> contarAnimalesPorGenero()`
 - `List<String> especiesConPeso (float pesoMin, float pesoMax)`
 - `Map<String, List<Animal>> agruparPorAlimentacion ()`
 - `DoubleSummaryStatistics resumenPesoConPelo (boolean tienePelo)`
 5. Paquete `clase`. Crear una clase `Alumno` con los atributos típicos de un alumno (nombre, fecha de nacimiento, dni, nia, teléfono, email, nota) y los correspondientes getters y setters (y constructor).
- Crea una clase `Aula` la cual contendrá un `List<Alumno>` con los alumnos que van a un aula determinada. Crea también los métodos siguientes:
- `int contarAlumnosMayoresQue (int edad)`

- float edadMedia ()
- List<String> nombreSuspendidos ()
- void listadoPorNiaDescendente ()
- boolean encontrarAlumnosDelMismoMesQue (Alumno a)

6. Paquete Olimpiada . A partir de las siguientes clases...

```
classDiagram
Participantes --o Disciplina
class Participantes{
- String dni
- String nombre
- String origen
- LocalDate fecha_nacimiento
- int disciplina
+ Participantes()
+ getters()
+ setters()
}
class Disciplina{
- int id
- String nombre
+ Disciplina()
+ getters()
+ setters()
}
```

se pide:

- Obtener la lista de todos los participantes junto con el nombre de la disciplina en la que participarán (los participantes de una misma disciplina se tienen que mostrar juntos).
- Obtener una lista de todas las disciplinas que no tengan participantes.
- Calcular la edad media de todos los participantes agrupada por disciplinas.
- Obtener las disciplinas con los participantes más joven y más viejo.

⌚19 de febrero de 2026

26. 8.3 Talleres

26.1. Taller UD07_01: GitHub Classroom

26.1.1. Unirnos a GitHub Classroom

Aceptamos el *Assignement* (la tarea/ejercicio) a partir del link del profesor, en este caso: <https://classroom.github.com/a/GjLK0ww7>

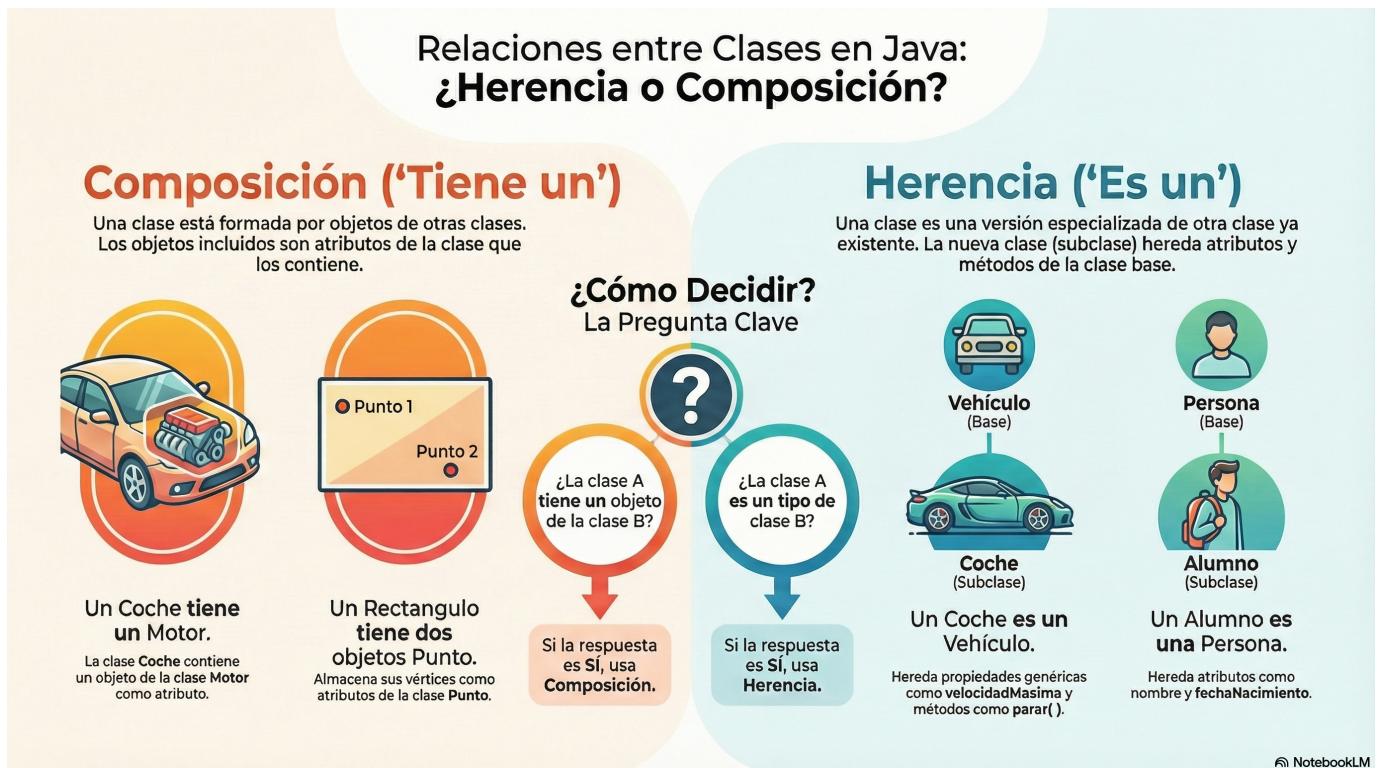
26.1.2. Tarea

Debes enviar tus soluciones a GitHub Classroom y superar al menos la mitad de los tests, cuantos más tests superados, mejor nota tendrás en la tarea.

 12 de febrero de 2026

9. UD08

27. 9.1 Composición, Herencia y Polimorfismo



27.1. Relaciones entre clases.

Cuando estudiaste el concepto de clase, ésta fue descrita como una especie de mecanismo de definición (plantillas), en el que se basaría el entorno de ejecución a la hora de construir un objeto: un mecanismo de definición de objetos.

Por tanto, a la hora de diseñar un conjunto de clases para modelar el conjunto de información cuyo tratamiento se desea automatizar, es importante establecer apropiadamente las posibles relaciones que puedan existir entre unas clases y otras.

En algunos casos es posible que no exista relación alguna entre unas clases y otras, pero lo más habitual es que sí exista: una clase puede ser una especialización (Relación entre dos clases donde una de ellas (la subclase) es una versión más especializada que la otra (la superclase), compartiendo características en común pero añadiendo ciertas características específicas que la especializan. El punto de vista inverso sería la generalización) de otra, o bien una generalización (Relación entre dos clases donde una de ellas (la superclase) es una versión más genérica que la otra (la subclase), compartiendo características en común pero sin las propiedades específicas que caracterizan a la subclase. El punto de vista inverso sería la especialización), o una clase contiene en su interior objetos de otra, o una clase utiliza a otra, etc. Es decir, que entre unas clases y otras habrá que definir cuál es su relación (si es que existe alguna).

Se pueden distinguir diversos tipos de relaciones entre clases:

- **Clientela.** Cuando una clase utiliza objetos de otra clase (por ejemplo al pasarlo como parámetros a través de un método).
- **Composición.** Cuando alguno de los atributos de una clase es un objeto de otra clase.
- **Anidamiento.** Cuando se definen clases en el interior de otra clase.
- **Herencia.** Cuando una clase comparte determinadas características con otra (clase base), añadiéndole alguna funcionalidad específica (especialización).

La relación de **clientela** la llevas utilizando desde que has empezado a programar en Java, pues desde tu clase principal (clase con método `main`) has estado declarando, creando y utilizando objetos de otras clases. Por ejemplo: si utilizas un objeto `String` dentro de la clase principal de tu programa, éste será cliente de la clase `String` (como sucederá con prácticamente cualquier

programa que se escriba en Java). Es la relación fundamental y más habitual entre clases (la utilización de unas clases por parte de otras) y, por supuesto, la que más vas a utilizar tú también, de hecho, ya la has estado utilizando y lo seguirás haciendo.

La relación de **composición** es posible que ya la hayas tenido en cuenta si has definido clases que contenían (tenían como atributos) otros objetos en su interior, lo cual es bastante habitual. Por ejemplo, si escribes una clase donde alguno de sus atributos es un objeto de tipo `String`, ya se está produciendo una relación de tipo composición (tu clase "tiene" un `String`, es decir, está compuesta por un objeto `String` y por algunos elementos más).

La relación de **anidamiento** (o anidación) es quizá menos habitual, pues implica declarar unas clases dentro de otras (clases internas o anidadas). En algunos casos puede resultar útil para tener un nivel más de encapsulamiento (ocultamiento del estado de un objeto, de sus datos miembro o atributos) de manera que sólo se puede cambiar mediante las operaciones (métodos) definidas para ese objeto. Cada objeto está aislado del exterior de manera que se protegen los datos contra su modificación por quien no tenga derecho a acceder a ellos, eliminando efectos secundarios y colaterales no deseados. Este modo de proceder permite que el usuario de una clase pueda obviar la implementación de los métodos y propiedades para concentrarse sólo en cómo usarlos. Por otro lado se evita que el usuario pueda cambiar su estado de manera imprevista e incontrolada y ocultación de información (efecto que se consigue gracias a la encapsulación: se evita la visibilidad de determinados miembros de una clase al resto del código del programa para de ese modo comunicarse con los objetos de la clase únicamente a través de su interfaz/métodos).

En el caso de la relación de **herencia** también la has visto ya, pues seguro que has utilizado unas clases que derivaban de otras, sobre todo, en el caso de los objetos que forman parte de las interfaces gráficas. Lo más probable es que hayas tenido que declarar clases que derivaban de algún componente gráfico (`JFrame`, `JDialog`, etc.).

Podría decirse que tanto la composición como la anidación son casos particulares de clientela, pues en realidad en todos esos casos una clase está haciendo uso de otra (al contener atributos que son objetos de la otra clase, al definir clases dentro de otras clases, al utilizar objetos en el paso de parámetros, al declarar variables locales utilizando otras clases, etc.).

A lo largo de la unidad, irás viendo distintas posibilidades de implementación de clases haciendo uso de todas estas relaciones, centrándonos especialmente en el caso de la herencia, que es la que permite establecer las relaciones más complejas.

27.1.1. Composición

Cuando en un sistema de información, una determinada entidad `A` contiene a otra `B` como una de sus partes, se suele decir que se está produciendo una relación de composición. Es decir, el objeto de la clase `A` contiene a uno o varios objetos de la clase `B`.

Por ejemplo, si describes una entidad `País` compuesta por una serie de atributos, entre los cuales se encuentra una lista de comunidades autónomas, podrías decir que los objetos de la clase `País` contienen varios objetos de la clase `ComunidadAutonoma`. Por otro lado, los objetos de la clase `ComunidadAutonoma` podrían contener como atributos objetos de la clase `Provincia`, la cual a su vez también podría contener objetos de la clase `Municipio`.

Como puedes observar, la composición puede encadenarse todas las veces que sea necesario hasta llegar a objetos básicos del lenguaje o hasta tipos primitivos que ya no contendrán otros objetos en su interior. Ésta es la forma más habitual de definir clases: mediante otras clases ya definidas anteriormente. Es una manera eficiente y sencilla de gestionar la reutilización de todo el código ya escrito. Si se definen clases que describen entidades distinguibles y con funciones claramente definidas, podrán utilizarse cada vez que haya que representar objetos similares dentro de otras clases.

Definición

La composición se da cuando una clase contiene algún atributo que es una referencia a un objeto de otra clase.

Una forma sencilla de plantearte si la relación que existe entre dos clases `A` y `B` es de composición podría ser mediante la expresión idiomática "tiene un": "la clase `A` tiene uno o varios objetos de la clase `B`", o visto de otro modo: "Objetos de la clase `B` pueden formar parte de la clase `A`".

Algunos ejemplos de composición podrían ser:

- Un coche tiene un motor y tiene cuatro ruedas.
- Una persona tiene un nombre, una fecha de nacimiento, una cuenta bancaria asociada para ingresar la nómina, etc.
- Un cocodrilo bajo investigación científica que tiene un número de dientes determinado, una edad, unas coordenadas de ubicación geográfica (medidas con GPS), etc.

Recuperando algunos de los ejemplos de clases que has utilizado en otras unidades:

- Una clase `Rectangulo` podría contener en su interior dos objetos de la clase `Punto` para almacenar los vértices inferior izquierdo y superior derecho.

- Una clase `Empleado` podría contener en su interior un objeto de la clase `DNI` para almacenar su DNI/NIF, y otro objeto de la clase `CuentaBancaria` para guardar la cuenta en la que se realizan los ingresos en nómina.



¿Podría decirse que la relación que existe entre la clase `Ave` y la clase `Loro` es una relación de composición?

No. Aunque claramente existe algún tipo de relación entre ambas, no parece que sea la de composición. No parece que se cumpla la expresión "tiene un": "Un loro tiene un ave". Se cumpliría más bien una expresión del tipo "es un": "Un loro es un ave". Algunos objetos que cumplirían la relación de composición podrían ser `Pico` o `Alas`, pues "un loro tiene un pico y dos alas", del mismo modo que "un ave tiene pico y dos alas". Este tipo de relación parece más de herencia (un loro es un tipo de ave).

27.1.2. Herencia

El mecanismo que permite crear clases basándose en otras que ya existen es conocido como herencia. Como ya has visto en unidades anteriores, Java implementa la herencia mediante la utilización de la palabra reservada `extends`.

El concepto de herencia es algo bastante simple y sin embargo muy potente: cuando se desea definir una nueva clase y ya existen clases que, de alguna manera, implementan parte de la funcionalidad que se necesita, es posible crear una nueva clase derivada de la que ya tienes. Al hacer esto se posibilita la reutilización de todos los atributos y métodos de la clase que se ha utilizado como base (clase padre o superclase), sin la necesidad de tener que escribirlos de nuevo.

Una subclase hereda todos los miembros de su clase padre (atributos, métodos y clases internas). Los constructores no se heredan, aunque se pueden invocar desde la subclase.

Algunos ejemplos de herencia podrían ser:

- Un coche es un vehículo (heredará atributos como la velocidad máxima o métodos como parar y arrancar).
- Un empleado es una persona (heredará atributos como el nombre o la fecha de nacimiento).
- Un rectángulo es una figura geométrica en el plano (heredará métodos como el cálculo de la superficie o de su perímetro).
- Un cocodrilo es un reptil (heredará atributos como por ejemplo el número de dientes).

En este caso la expresión idiomática que puedes usar para plantearte si el tipo de relación entre dos clases A y B es de herencia podría ser "es un": "la clase A es un tipo específico de la clase B" (especialización), o visto de otro modo: "la clase B es un caso general de la clase A" (generalización).

Recuperando algunos ejemplos de clases que ya has utilizado en otras unidades:

- Una ventana en una aplicación gráfica puede ser una clase que herede de `JFrame` (componente `Swing`: `javax.swing.JFrame`), de esta manera esa clase será un marco que dispondrá de todos los métodos y atributos de `JFrame` mas aquéllos que tú decidas incorporarle al rellenarlo de componentes gráficos.
- Una caja de diálogo puede ser un tipo de `JDialog` (otro componente `Swing`: `javax.swing.JDialog`).

En Java, la clase `Object` (dentro del paquete `java.lang`) define e implementa el comportamiento común a todas las clases (incluidas aquellas que tú escribas). Como recordarás, ya se dijo que en Java cualquier clase deriva en última instancia de la clase `Object`.

Todas las clases tienen una clase padre, que a su vez también posee una superclase, y así sucesivamente hasta llegar a la clase `Object`. De esta manera, se construye lo que habitualmente se conoce como una jerarquía de clases, que en el caso de Java tendría a la clase `Object` en la raíz.



Atención

Cuando escribes una clase en Java, puedes hacer que herede de una determinada clase padre (mediante el uso de `extends`) o bien no indicar ninguna herencia. En tal caso, aunque no indiques explícitamente ningún tipo de herencia, el compilador asumirá entonces de manera implícita que tu clase hereda de la clase `Object`, que define e implementa el comportamiento común a todas las clases.

27.1.3. ¿Herencia o composición?

Cuando escribes tus propias clases, debes intentar tener claro en qué casos utilizar la composición y cuándo la herencia:

- **Composición:** cuando una clase está formada por objetos de otras clases. En estos casos se incluyen objetos de esas clases, pero no necesariamente se comparten características con ellos (no se heredan características de esos objetos, sino que directamente se utilizarán sus atributos y sus métodos). Esos objetos incluidos no son más que atributos miembros de la clase que se está definiendo.

- **Herencia:** cuando una clase cumple todas las características de otra. En estos casos la clase derivada es una especialización (o particularización, extensión o restricción) de la clase base. Desde otro punto de vista se diría que la clase base es una generalización de las clases derivadas.

Por ejemplo, imagina que dispones de una clase Punto (ya la has utilizado en otras ocasiones) y decides definir una nueva clase llamada Círculo. Dado que un punto tiene como atributos sus coordenadas en plano (x_1, y_1), decides que es buena idea aprovechar esa información e incorporarla en la clase Circulo que estás escribiendo. Para ello utilizas la herencia, de manera que al derivar la clase Círculo de la clase Punto, tendrás disponibles los atributos x_1 e y_1 . Ahora solo faltaría añadirle algunos atributos y métodos más como por ejemplo el radio del círculo, el cálculo de su área y su perímetro, etc.

En principio parece que la idea pueda funcionar pero es posible que más adelante, si continúas construyendo una jerarquía de clases, observes que puedas llegar a conclusiones incongruentes al suponer que un círculo es una especialización de un punto (un tipo de punto). ¿Todas aquellas figuras que contengan uno o varios puntos deberían ser tipos de punto? ¿Y si tienes varios puntos? ¿Cómo accedes a ellos? ¿Un rectángulo también tiene sentido que herede de un punto? No parece muy buena idea.

Parece que en este caso habría resultado mejor establecer una relación de composición. Analízalo detenidamente: ¿cuál de estas dos situaciones te suena mejor? 1. "Un círculo es un punto (su centro)", y por tanto heredará las coordenadas x_1 e y_1 que tiene todo punto. Además tendrá otras características específicas como el radio o métodos como el cálculo de la longitud de su perímetro o de su área. 2. "Un círculo tiene un punto (su centro)", junto con algunos atributos más como por ejemplo el radio. También tendrá métodos para el cálculo de su área o de la longitud de su perímetro.

Parece que en este caso la composición refleja con mayor fidelidad la relación que existe entre ambas clases. Normalmente suele ser suficiente con plantearse las preguntas "¿A es un tipo de B?" o "¿A contiene elementos de tipo B?".

27.2. Composición

27.2.1. Sintaxis de la composición.

Para indicar que una clase contiene objetos de otra clase no es necesaria ninguna sintaxis especial. Cada uno de esos objetos no es más que un atributo y, por tanto, debe ser declarado como tal:

```

1  class <nombreClase> {
2      [modificadores] <NombreClase1> nombreAtributo1;
3      [modificadores] <NombreClase2> nombreAtributo2;
4      <NombreClase3>[] listado;
5
6 }
```

En unidades anteriores has trabajado con la clase `Punto`, que definía las coordenadas de un punto en el plano, y con la clase `Rectangulo`, que definía una figura de tipo rectángulo también en el plano a partir de dos de sus vértices (inferior izquierdo y superior derecho). Tal y como hemos formalizado ahora los tipos de relaciones entre clases, parece bastante claro que aquí tendrías un caso de composición: "un rectángulo contiene puntos". Por tanto, podrías ahora redefinir los atributos de la clase `Rectangulo` (cuatro números reales) como dos objetos de tipo `Punto`:

```

1  class Rectangulo {
2      private Punto vertice1;
3      private Punto vertice2;
4      ...
5 }
```

Ahora los métodos de esta clase deberán tener en cuenta que ya no hay cuatro atributos de tipo `double`, sino dos atributos de tipo `Punto` (cada uno de los cuales contendrá en su interior dos atributos de tipo `double`).

Revisa con cuidado el [Ejemplo 2.1](#)

27.2.2. Uso de la composición (I). Preservación de la ocultación.

Como ya has observado, la relación de composición no tiene más misterio a la hora de implementarse que simplemente declarar atributos de las clases que necesites dentro de la clase que estés diseñando.

Ahora bien, cuando escribes clases que contienen objetos de otras clases (lo cual será lo más habitual) deberás tener un poco de precaución con aquellos métodos que devuelvan información acerca de los atributos de la clase (métodos "consultores" o de tipo `get`).

Como ya viste en la unidad dedicada a la creación de clases, lo normal suele ser declarar los atributos como privados (o protegidos, como veremos un poco más adelante) para ocultarlos a los posibles clientes de la clase (otros objetos que en el futuro harán uso de la clase). Para que otros objetos puedan acceder a la información contenida en los atributos, o al menos a una parte de ella, deberán hacerlo a través de métodos que sirvan de interfaz, de manera que sólo se podrá tener acceso a aquella

información que el creador de la clase haya considerado oportuna. Del mismo modo, los atributos solamente serán modificados desde los métodos de la clase, que decidirán cómo y bajo qué circunstancias deben realizarse esas modificaciones. Con esa metodología de acceso se tenía perfectamente separada la parte de manipulación interna de los atributos de la interfaz con el exterior.

Hasta ahora los métodos de tipo get devolvían tipos primitivos, es decir, copias del contenido (a veces con algún tipo de modificación o de formato) que había almacenado en los atributos, pero los atributos seguían "a salvo" como elementos privados de la clase. Pero, a partir de este momento, al tener objetos dentro de las clases y no sólo tipos primitivos, es posible que en un determinado momento interese devolver un objeto completo.

Ahora bien, cuando vayas a devolver un objeto habrás de obrar con mucha precaución. Si en un método de la clase devuelves directamente un objeto que es un atributo, estarás ofreciendo directamente una referencia a un objeto atributo que probablemente has definido como privado. ¡De esta forma estás volviendo a hacer público un atributo que inicialmente era privado!

Para evitar ese tipo de situaciones (ofrecer al exterior referencias a objetos privados) puedes optar por diversas alternativas, procurando siempre evitar la devolución directa de un atributo que sea un objeto:

- Una opción podría ser devolver siempre tipos primitivos.
- Dado que esto no siempre es posible, o como mínimo poco práctico, otra posibilidad es crear un nuevo objeto que sea una copia del atributo que quieras devolver y utilizar ese objeto como valor de retorno. Es decir, crear una copia del objeto especialmente para devolverlo. De esta manera, el código cliente de ese método podrá manipular a su antojo ese nuevo objeto, pues no será una referencia al atributo original, sino un nuevo objeto con el mismo contenido.

Por último, debes tener en cuenta que es posible que en algunos casos sí se necesite realmente la referencia al atributo original (algo muy habitual en el caso de atributos estáticos). En tales casos, no habrá problema en devolver directamente el atributo para que el código llamante (cliente) haga el uso que estime oportuno de él.



Devolución de objetos

Debes evitar por todos los medios la devolución de un atributo que sea un objeto (estarías dando directamente una referencia al atributo, visible y manipulable desde fuera), salvo que se trate de un caso en el que deba ser así.

Para entender estas situaciones un poco mejor, podemos volver a la clase `Rectangulo` y observar sus nuevos métodos de tipo get.

Revisa con cuidado el [Ejemplo 2.2](#)

27.2.3. Uso de la composición (II). Llamadas a constructores.

Otro factor que debes considerar, a la hora de escribir clases que contengan como atributos objetos de otras clases, es su comportamiento a la hora de instanciarse. Durante el proceso de creación de un objeto (constructor) de la clase contenedora habrá que tener en cuenta también la creación (llamadas a constructores) de aquellos objetos que son contenidos.



El constructor de la clase contenedora debe invocar a los constructores de las clases de los objetos contenidos.

En este caso hay que tener cuidado con las referencias a objetos que se pasan como parámetros para llenar el contenido de los atributos. Es conveniente hacer una copia de esos objetos y utilizar esas copias para los atributos pues si se utiliza la referencia que se ha pasado como parámetro, el código cliente de la clase podría tener acceso a ella sin necesidad de pasar por la interfaz de la clase (volveríamos a dejar abierta una puerta pública a algo que quizás sea privado).

Además, si el objeto parámetro que se pasó al constructor formaba parte de otro objeto, esto podría ocasionar un desagradable efecto colateral si esos objetos son modificados en el futuro desde el código cliente de la clase, ya que no sabes de dónde provienen esos objetos, si fueron creados especialmente para ser usados por el nuevo objeto creado o si pertenecen a otro objeto que podría modificarlos más tarde. Es decir, correrías el riesgo de estar "compartiendo" esos objetos con otras partes del código, sin ningún tipo de control de acceso y con las nefastas consecuencias que eso podría tener: cualquier cambio de ese objeto afectaría a partes del programa supuestamente independientes, que entienden ese objeto como suyo.

Acción

En el fondo los objetos no son más que variables de tipo referencia a la zona de memoria en la que se encuentra toda la información del objeto en sí mismo. Esto es, puedes tener un único objeto y múltiples referencias a él. Pero sólo se trata de un objeto, y cualquier modificación desde una de sus referencias afectaría a todas las demás, pues estamos hablando del mismo objeto.

Recuerda también que sólo se crean objetos cuando se llama a un constructor (uso de new). Si realizas asignaciones o pasos de parámetros, no se están copiando o pasando copias de los objetos, sino simplemente de las referencias, y por tanto se tratará siempre del mismo objeto.

Se trata de un efecto similar al que sucedía en los métodos de tipo get, pero en este caso en sentido contrario (en lugar de que nuestra clase "regale" al exterior uno de sus atributos objeto mediante una referencia, en esta ocasión se "adueña" de un parámetro objeto que probablemente pertenezca a otro objeto y que es posible que el futuro haga uso de él).

Para entender mejor estos posibles efectos podemos continuar con el ejemplo de la clase `Rectangulo` que contiene en su interior dos objetos de la clase `Punto`. En los constructores del rectángulo habrá que incluir todo lo necesario para crear dos instancias de la clase `Punto` evitando las referencias a parámetros (haciendo copias).

Revisa con cuidado el [Ejemplo 2.2.1](#)

27.2.4. Clases anidadas o internas.

En algunos lenguajes, es posible definir una clase dentro de otra clase (clases internas):

```

1  class ClaseContenedora {
2      // Cuerpo de la clase
3      ...
4      class ClaseInterna {
5          // Cuerpo de la clase interna
6          ...
7      }
8 }
```

Se pueden distinguir varios tipos de clases internas:

- Clases internas estáticas (o clases anidadas), declaradas con el modificador `static`.
- Clases internas miembro, conocidas habitualmente como clases internas. Declaradas al máximo nivel de la clase contenedora y no estáticas.
- Clases internas locales, que se declaran en el interior de un bloque de código (normalmente dentro de un método).
- Clases anónimas, similares a las internas locales, pero sin nombre (sólo existirá un objeto de ellas y, al no tener nombre, no tendrán constructores). Se suelen usar en la gestión de eventos en los interfaces gráficos.
- Aquí tienes algunos ejemplos:

```

1  class ClaseContenedora {
2      ...
3      static class ClaseAnidadaEstatica {
4          ...
5      }
6      class ClaseInterna {
7          ...
8      }
9 }
```

Las clases anidadas, como miembros de una clase que son (miembros de `ClaseContenedora`), pueden ser declaradas con los modificadores `public`, `protected`, `private` o de paquete, como el resto de miembros.

Las clases internas (no estáticas) tienen acceso a otros miembros de la clase dentro de la que está definida aunque sean privados (se trata en cierto modo de un miembro más de la clase), mientras que las anidadas (estáticas) no.

Las clases internas se utilizan en algunos casos para:

- Agrupar clases que sólo tiene sentido que existan en el entorno de la clase en la que han sido definidas, de manera que se oculta su existencia al resto del código.
- Incrementar el nivel de encapsulación y ocultamiento.
- Proporcionar un código fuente más legible y fácil de mantener (el código de las clases internas y anidadas está más cerca de donde es usado).

En Java es posible definir clases internas y anidadas, permitiendo todas esas posibilidades. Aunque para los ejemplos con los que vas a trabajar no las vas a necesitar por ahora.

27.3. Herencia

Como ya has estudiado, la herencia es el mecanismo que permite definir una nueva clase a partir de otra, pudiendo añadir nuevas características, sin tener que volver a escribir todo el código de la clase base.

La clase de la que se hereda suele ser llamada clase base, clase padre o superclase (de la que hereda otra clase. Se heredan todas aquellas características que la clase padre permita). A la clase que hereda se le suele llamar clase hija, clase derivada o subclase (que hereda de otra clase. Se heredan todas aquellas características que la clase padre permita).

Una clase derivada puede ser a su vez clase padre de otra que herede de ella y así sucesivamente dando lugar a una jerarquía de clases, excepto aquellas que estén en la parte de arriba de la jerarquía (sólo serán clases padre) o en la parte de abajo (sólo serán clases hijas).

Una clase hija no tiene acceso a los miembros privados de su clase padre, tan solo a los públicos (como cualquier parte del código tendría) y los protegidos (a los que sólo tienen acceso las clases derivadas y las del mismo paquete). Aquellos miembros que sean privados en la clase base también habrán sido heredados, pero el acceso a ellos estará restringido al propio funcionamiento de la superclase y sólo se podrá acceder a ellos si la superclase ha dejado algún medio indirecto para hacerlo (por ejemplo a través de algún método).

Todos los miembros de la superclase, tanto atributos como métodos, son heredados por la subclase. Algunos de estos miembros heredados podrán ser redefinidos o sobreescritos (overridden) y también podrán añadirse nuevos miembros. De alguna manera podría decirse que estás "ampliando" la clase base con características adicionales o modificando algunas de ellas (proceso de especialización).

Recuerda

Una clase derivada extiende la funcionalidad de la clase base sin tener que volver a escribir el código de la clase base.

27.3.1. Sintaxis de la herencia.

En Java la herencia se indica mediante la palabra reservada `extends`:

```

1 [modificador] class ClasePadre {
2     // Cuerpo de la clase
3     ...
4 }
5
6 [modificador] class ClaseHija extends ClasePadre {
7     // Cuerpo de la clase
8     ...
9 }
```

Imagina que tienes una clase `Persona` que contiene atributos como `nombre`, `apellidos` y `fecha de nacimiento`:

```

1 public class Persona {
2     String nombre;
3     String apellidos;
4     LocalDate fechaNacim;
5     ...
6 }
```

Es posible que, más adelante, necesites una clase `Alumno` que compartirá esos atributos (dado que todo alumno es una persona, pero con algunas características específicas que lo especializan). En tal caso tendrías la posibilidad de crear una clase `Alumno` que repitiera todos esos atributos o bien heredar de la clase `Persona`:

```

1 public class Alumno extends Persona {
2     String grupo;
3     double notaMedia;
4     ...
5 }
```

A partir de ahora, un objeto de la clase `Alumno` contendrá los atributos `grupo` y `notaMedia` (propios de la clase `Alumno`), pero también `nombre`, `apellidos` y `fechaNacim` (propios de su clase base `Persona` y que por tanto ha heredado).

Revisa con cuidado el [Ejemplo 3.1](#)

27.3.2. Acceso a miembros heredados.

Como ya has visto anteriormente, no es posible acceder a miembros privados de una superclase. Para poder acceder a ellos podrías pensar en hacerlos públicos, pero entonces estarías dando la opción de acceder a ellos a cualquier objeto externo y es probable que tampoco sea eso lo deseable. Para ello se inventó el modificador `protected` (protegido) que permite el acceso desde clases heredadas, pero no desde fuera de las clases (estrictamente hablando, desde fuera del paquete), que serían como miembros privados.

En la unidad dedicada a la utilización de clases ya estudiaste los posibles modificadores de acceso que podía tener un miembro: sin modificador (acceso de paquete), público, privado o protegido.

Aquí tienes de nuevo el resumen:

modificador	Misma clase	Mismo paquete	Subclase	Otro paquete
<code>public</code>	✓	✓	✓	✓
<code>protected</code>	✓	✓	✓	✗
Sin modificador (package)	✓	✓	✗	✗
<code>private</code>	✓	✗	✗	✗

⚠ Recuerda

¡Recuerda que **los modificadores de acceso son excluyentes!** Sólo se puede utilizar uno de ellos en la declaración de un atributo.

Si en el ejemplo anterior de la clase `Persona` se hubieran definido sus atributos como `private`:

```
1 public class Persona {
2     private String nombre;
3     private String apellidos;
4     ...
5 }
```

Al definir la clase `Alumno` como heredera de `Persona`, no habrías tenido acceso a esos atributos, pudiendo ocasionar un grave problema de operatividad al intentar manipular esa información. Por tanto, en estos casos lo más recomendable habría sido declarar esos atributos como `protected` o bien sin modificador (para que también tengan acceso a ellos otras clases del mismo paquete, si es que se considera oportuno):

```
1 public class Persona {
2     protected String nombre;
3     protected String apellidos;
4     ...
5 }
```

💡 private vs protected

Sólo en aquellos casos en los que se desea explícitamente que un miembro de una clase no pueda ser accesible desde una clase derivada debería utilizarse el modificador `private`. En el resto de casos es recomendable utilizar `protected`, o bien no indicar modificador (acceso a nivel de paquete).

Revisa con cuidado el [Ejemplo 3.2](#)

27.3.3. Utilización de miembros heredados (I). Atributos.

Los atributos heredados por una clase son, a efectos prácticos, iguales que aquellos que sean definidos específicamente en la nueva clase derivada.

En el ejemplo anterior la clase `Persona` disponía de tres atributos y la clase `Alumno`, que heredaba de ella, añadía dos atributos más. Desde un punto de vista funcional podrías considerar que la clase `Alumno` tiene cinco atributos: tres por ser `Persona` (nombre, apellidos, fecha de nacimiento) y otros dos más por ser `Alumno` (grupo y nota media).

Revisa con cuidado el [Ejemplo 3.3](#)

27.3.4. Utilización de miembros heredados (II). Métodos.

Del mismo modo que se heredan los atributos, también se heredan los métodos, convirtiéndose a partir de ese momento en otros métodos más de la clase derivada, junto a los que hayan sido definidos específicamente.

En el ejemplo de la clase `Persona`, si dispusiéramos de métodos `get` y `set` para cada uno de sus tres atributos (`nombre`, `apellidos`, `fechaNacim`), tendrías seis métodos que podrían ser heredados por sus clases derivadas. Podrías decir entonces que la clase `Alumno`, derivada de `Persona`, tiene diez métodos:

- Seis por ser `Persona` (`getNombre`, `getApellidos`, `getFechaNacim`, `setNombre`, `setApellidos`, `setFechaNacim`).
- Oros cuatro más por ser `Alumno` (`getGrupo`, `setGrupo`, `getNotaMedia`, `setNotaMedia`).

Sin embargo, sólo tendrías que definir esos cuatro últimos (los específicos) pues los genéricos ya los has heredado de la superclase.

Revisa con cuidado el [Ejemplo 3.3.1](#)

27.3.5. Redefinición de métodos heredados.

Una clase puede redefinir algunos de los métodos que ha heredado de su clase base. En tal caso, el nuevo método (especializado) sustituye al heredado. Este procedimiento también es conocido como de sobrescritura de métodos.

En cualquier caso, aunque un método sea sobreescrito o redefinido, aún es posible acceder a él a través de la referencia super, aunque sólo se podrá acceder a métodos de la clase padre y no a métodos de clases superiores en la jerarquía de herencia.

Los métodos redefinidos pueden ampliar su accesibilidad con respecto a la que ofrece el método original de la superclase, pero nunca restringirla. Por ejemplo, si un método es declarado como `protected` o de paquete en la clase base, podría ser redefinido como `public` en una clase derivada. Los métodos estáticos o de clase no pueden ser sobreescritos. Los originales de la clase base permanecen inalterables a través de toda la jerarquía de herencia.

En el ejemplo de la clase `Alumno`, podrían redefinirse algunos de los métodos heredados. Por ejemplo, imagina que el método `getApellidos` devuelva la cadena "Alumno:" junto con los apellidos del alumno. En tal caso habría que escribir ese método para realizar esa modificación:

```
1 public String getApellidos () {
2     return "Alumno: " + apellidos;
3 }
```

Cuando sobrescribas un método heredado en Java puedes incluir la anotación `@Override`. Esto indicará al compilador que tu intención es sobreescibir el método de la clase padre. De este modo, si te equivocas (por ejemplo, al escribir el nombre del método) y no lo estás realmente sobreescribiendo, el compilador producirá un error y así podrás darte cuenta del fallo. En cualquier caso, no es necesario indicar `@Override`, pero puede resultar de ayuda a la hora de localizar este tipo de errores (crees que has sobreescrito un método heredado y al confundirte en una letra estás realmente creando un nuevo método diferente). En el caso del ejemplo anterior quedaría:

```
1 @Override
2 public String getApellidos ()
```

Revisa con cuidado el [Ejemplo 3.4](#)

27.3.6. Ampliación de métodos heredados.

Hasta ahora, has visto que para redefinir o sustituir un método de una superclase es suficiente con crear otro método en la subclase que tenga el mismo nombre que el método que se desea sobreescibir. Pero, en otras ocasiones, puede que lo que necesites no sea sustituir completamente el comportamiento del método de la superclase, sino simplemente ampliarlo.

Para poder hacer esto necesitas poder preservar el comportamiento antiguo (el de la superclase) y añadir el nuevo (el de la subclase). Para ello, puedes invocar desde el método "ampliador" de la clase derivada al método "ampliado" de la clase superior (teniendo ambos métodos el mismo nombre). ¿Cómo se puede conseguir eso? Puedes hacerlo mediante el uso de la referencia `super`.

La palabra reservada `super` es una referencia a la clase padre de la clase en la que te encuentres en cada momento (es algo similar a `this`, que representaba una referencia a la clase actual). De esta manera, podrías invocar a cualquier método de tu superclase (si es que se tiene acceso a él).

Por ejemplo, imagina que la clase `Persona` dispone de un método que permite mostrar el contenido de algunos datos personales de los objetos de este tipo (nombre, apellidos, etc.). Por otro lado, la clase `Alumno` también necesita un método similar, pero que

muestre también su información especializada (grupo, nota media, etc.). ¿Cómo podrías aprovechar el método de la superclase para no tener que volver a escribir su contenido en la subclase?

Podría hacerse de una manera tan sencilla como la siguiente:

```

1  public void mostrar () {
2      super.mostrar ();
3      // Llamada al método "mostrar" de la superclase
4      // A continuación mostramos la información "especializada" de esta subclase
5      System.out.printf ("Grupo: %s\n", this.grupo);
6      System.out.printf ("Nota media: %.2f\n", this.notaMedia);
7  }

```

Este tipo de ampliaciones de métodos resultan especialmente útiles por ejemplo en el caso de los constructores, donde se podría ir llamando a los constructores de cada superclase encadenadamente hasta el constructor de la clase en la cúspide de la jerarquía (el constructor de la clase `Object`).

Revisa con cuidado el [Ejemplo 3.5](#)

27.3.7. Constructores y herencia.

Recuerda que cuando estudiaste los constructores viste que un constructor de una clase puede llamar a otro constructor de la misma clase, previamente definido, a través de la referencia `this`. En estos casos, la utilización de `this` sólo podía hacerse en la primera línea de código del constructor.

Como ya has visto, un constructor de una clase derivada puede hacer algo parecido para llamar al constructor de su clase base mediante el uso de la palabra `super`. De esta manera, el constructor de una clase derivada puede llamar primero al constructor de su superclase para que inicialice los atributos heredados y posteriormente se inicializarán los atributos específicos de la clase: los no heredados. Nuevamente, esta llamada también debe ser la primera sentencia de un constructor (con la única excepción de que exista una llamada a otro constructor de la clase mediante `this`).

Si no se incluye una llamada a `super()` dentro del constructor, el compilador incluye automáticamente una llamada al constructor por defecto de clase base (llamada a `super()`). Esto da lugar a una llamada en cadena de constructores de superclase hasta llegar a la clase más alta de la jerarquía (que en Java es la clase `Object`).

En el caso del constructor por defecto (el que crea el compilador si el programador no ha escrito ninguno), el compilador añade lo primero de todo, antes de la inicialización de los atributos a sus valores por defecto, una llamada al constructor de la clase base mediante la referencia `super`.

A la hora de destruir un objeto (método `finalize()`) es importante llamar a los finalizadores en el orden inverso a como fueron llamados los constructores (primero se liberan los recursos de la clase derivada y después los de la clase base mediante la llamada `super.finalize()`).

Si la clase `Persona` tuviera un constructor de este tipo:

```

1  public Persona (String nombre, String apellidos, LocalDate fechaNacim) {
2      this.nombre= nombre;
3      this.apellidos= apellidos;
4      this.fechaNacim= new LocalDate (fechaNacim);
5  }

```

Podrías llamarlo desde un constructor de una clase derivada (por ejemplo `Alumno`) de la siguiente forma:

```

1  public Alumno (String nombre, String apellidos, LocalDate fechaNacim, String grupo, double notaMedia) {
2      super (nombre, apellidos, fechaNacim);
3      this.grupo= grupo;
4      this.notaMedia= notaMedia;
5  }

```

En realidad se trata de otro recurso más para optimizar la reutilización de código, en este caso el del constructor, que aunque no es heredado, sí puedes invocarlo para no tener que reescribirlo.

Revisa con cuidado el [Ejemplo 3.6](#)

27.3.8. Creación y utilización de clases derivadas.

Ya has visto cómo crear una clase derivada, cómo acceder a los miembros heredados de las clases superiores, cómo redefinir algunos de ellos e incluso cómo invocar a un constructor de la superclase. Ahora se trata de poner en práctica todo lo que has

aprendido para que puedas crear tus propias jerarquías de clases, o basarte en clases que ya existan en Java para heredar de ellas, y las utilices de manera adecuada para que tus aplicaciones sean más fáciles de escribir y mantener.

Herencia = menos código

La idea de la herencia no es complicar los programas, sino todo lo contrario: simplificarlos al máximo. Procurar que haya que escribir la menor cantidad posible de código repetitivo e intentar facilitar en lo posible la realización de cambios (bien para corregir errores bien para incrementar la funcionalidad).

27.3.9. La clase `Object` en Java.

Todas las clases en Java son descendentes (directos o indirectos) de la clase Object. Esta clase define los estados y comportamientos básicos que deben tener todos los objetos. Entre estos comportamientos, se encuentran:

- La posibilidad de compararse.
- La capacidad de convertirse a cadenas.
- La habilidad de devolver la clase del objeto.

Entre los métodos que incorpora la clase `Object` y que por tanto hereda cualquier clase en Java tienes:

Principales métodos de la clase `Object`:

Método	Descripción
<code>Object()</code>	Constructor.
<code>clone()</code>	Método clonador: crea y devuelve una copia del objeto ("clona" el objeto).
<code>boolean equals(Object obj)</code>	Indica si el objeto pasado como parámetro es igual a este objeto.
<code>void finalize()</code>	Método llamado por el recolector de basura cuando éste considera que no queda ninguna referencia a este objeto en el entorno de ejecución.
<code>int hashCode()</code>	Devuelve un código hash para el objeto.
<code>toString()</code>	Devuelve una representación del objeto en forma de String.

La clase `Object` representa la superclase que se encuentra en la cúspide de la jerarquía de herencia en Java. Cualquier clase (incluso las que tú implementes) acaban heredando de ella.

27.3.10. Herencia múltiple.

En determinados casos podrías considerar la posibilidad de que se necesite heredar de más de una clase, para así disponer de los miembros de dos (o más) clases disjuntas (que no derivan una de la otra). La herencia múltiple permite hacer eso: recoger las distintas características (atributos y métodos) de clases diferentes formando una nueva clase derivada de varias clases base.

El problema en estos casos es la posibilidad que existe de que se produzcan ambigüedades, así, si tuviéramos miembros con el mismo identificador en clases base diferentes, en tal caso, ¿qué miembro se hereda? Para evitar esto, los compiladores suelen solicitar que ante casos de ambigüedad, se especifique de manera explícita la clase de la cual se quiere utilizar un determinado miembro que pueda ser ambiguo.

Ahora bien, la posibilidad de herencia múltiple no está disponible en todos los lenguajes orientados a objetos, ¿lo estará en Java? La respuesta es **negativa**.



Herencia múltiple

En Java **no existe** la herencia múltiple de clases.

27.4. Clases Abstractas

En determinadas ocasiones, es posible que necesites definir una clase que represente un concepto lo suficientemente abstracto como para que nunca vayan a existir instancias de ella (objetos). ¿Tendría eso sentido? ¿Qué utilidad podría tener?

Imagina una aplicación para un centro educativo que utilice las clases de ejemplo Alumno y Profesor, ambas subclases de Persona. Es más que probable que esa aplicación nunca llegue a necesitar objetos de la clase Persona, pues serían demasiado genéricos como para poder ser utilizados (no contendrían suficiente información específica). Podrías llegar entonces a la conclusión de que la clase Persona ha resultado de utilidad como clase base para construir otras clases que hereden de ella, pero no como una clase instanciable de la cual vayan a existir objetos. A este tipo de clases se les llama **clases abstractas**.

Clases abstractas

En algunos casos puede resultar útil disponer de clases que nunca serán instanciadas, sino que proporcionan un marco o modelo a seguir por sus clases derivadas dentro de una jerarquía de herencia. Son las **clases abstractas**.

La posibilidad de declarar clases abstractas es una de las características más útiles de los lenguajes orientados a objetos, pues permiten dar unas líneas generales de cómo es una clase sin tener que implementar todos sus métodos o implementando solamente algunos de ellos. Esto resulta especialmente útil cuando las distintas clases derivadas deban proporcionar los mismos métodos indicados en la clase base abstracta, pero su implementación sea específica para cada subclase.

Imagina que estás trabajando en un entorno de manipulación de objetos gráficos y necesitas trabajar con líneas, círculos, rectángulos, etc. Estos objetos tendrán en común algunos atributos que representen su estado (ubicación, color del contorno, color de relleno, etc.) y algunos métodos que modelen su comportamiento (dibujar, llenar con un color, escalar, desplazar, rotar, etc.). Algunos de ellos serán comunes para todos ellos (por ejemplo la ubicación o el desplazamiento) y sin embargo otros (como por ejemplo dibujar) necesitarán una implementación específica dependiendo del tipo de objeto. Pero, en cualquier caso, todos ellos necesitan esos métodos (tanto un círculo como un rectángulo necesitan el método dibujar, aunque se lleven a cabo de manera diferente). En este caso resultaría muy útil disponer de una clase abstracta objeto gráfico donde se definirían las líneas generales (algunos atributos concretos comunes, algunos métodos concretos comunes implementados y algunos métodos genéricos comunes sin implementar) de un objeto gráfico y más adelante, según se vayan definiendo clases especializadas (líneas, círculos, rectángulos), se irán concretando en cada subclase aquellos métodos que se dejaron sin implementar en la clase abstracta.

27.4.1. Declaración de una clase abstracta.

Ya has visto que una clase abstracta es una clase que no se puede instanciar, es decir, que no se pueden crear objetos a partir de ella. La idea es permitir que otras clases deriven de ella, proporcionando un modelo genérico y algunos métodos de utilidad general. Las clases abstractas se declaran mediante el modificador `abstract`:

```
1 [modificador_acceso] abstract class nombreClase [herencia] [interfaces] {
2 ...
3 }
```

Métodos abstractos

Una clase puede contener en su interior métodos declarados como `abstract` (métodos para los cuales sólo se indica la cabecera, pero no se proporciona su implementación). En tal caso, la clase tendrá que ser necesariamente también `abstract`. Esos métodos tendrán que ser posteriormente implementados en sus clases derivadas.

Por otro lado, una clase también puede contener métodos totalmente implementados (no abstractos), los cuales serán heredados por sus clases derivadas y podrán ser utilizados sin necesidad de definirlos (pues ya están implementados).

Cuando trabajes con clases abstractas debes tener en cuenta:

- Una clase abstracta sólo puede usarse para crear nuevas clases derivadas. No se puede hacer un `new` de una clase abstracta. Se produciría un error de compilación.
- Una clase abstracta puede contener métodos totalmente definidos (no abstractos) y métodos sin definir (métodos abstractos).

Revisa con cuidado el [Ejemplo 4.1](#)

27.4.2. Métodos abstractos.

Un método abstracto es un método declarado en una clase para el cual esa clase no proporciona la implementación. Si una clase dispone de al menos un método abstracto se dice que es una clase abstracta. Toda clase que herede (sea subclase) de una clase abstracta debe implementar todos los métodos abstractos de su superclase o bien volverlos a declarar como abstractos (y por tanto también sería abstracta). Para declarar un método abstracto en Java se utiliza el modificador `abstract` (es un método cuya implementación no se define, sino que se declara únicamente su interfaz (cabecera) para que su cuerpo sea implementado más adelante en una clase derivada).

Un método se declara como abstracto mediante el uso del modificador `abstract` (como en las clases abstractas):

```
1 [modificador_acceso] abstract <tipo> <nombreMetodo> ([parámetros]) [excepciones];
```

Estos métodos tendrán que ser obligatoriamente redefinidos (en realidad "definidos", pues aún no tienen contenido) en las clases derivadas. Si en una clase derivada se deja algún método abstracto sin implementar, esa clase derivada será también una clase abstracta.

Clase abstracta?

Cuando una clase contiene un método abstracto tiene que declararse como abstracta obligatoriamente.

Imagina que tienes una clase `Empleado` genérica para diversos tipos de empleado y tres clases derivadas: `EmpleadoFijo` (tiene un salario fijo más ciertos complementos), `EmpleadoTemporal` (salario fijo más otros complementos diferentes) y `EmpleadoComercial` (una parte de salario fijo y unas comisiones por cada operación). La clase `Empleado` podría contener un método abstracto `calcularNomina`, pues sabes que se método será necesario para cualquier tipo de empleado (todo empleado cobra una nómina). Sin embargo el cálculo en sí de la nómina será diferente si se trata de un empleado fijo, un empleado temporal o un empleado comercial, y será dentro de las clases especializadas de `Empleado` (`EmpleadoFijo`, `EmpleadoTemporal`, `EmpleadoComercial`) donde se implementen de manera específica el cálculo de las mismas.

Debes tener en cuenta al trabajar con métodos abstractos:

- Un método abstracto implica que la clase a la que pertenece tiene que ser abstracta, pero eso no significa que todos los métodos de esa clase tengan que ser abstractos.
- Un método abstracto no puede ser privado (no se podría implementar, dado que las clases derivadas no tendrían acceso a él).
- Los métodos abstractos no pueden ser estáticos, pues los métodos estáticos no pueden ser redefinidos (y los métodos abstractos necesitan ser redefinidos).

Revisa con cuidado el [Ejemplo 4.2](#)

27.4.3. Clases y métodos finales.

En unidades anteriores has visto el modificador `final`, aunque sólo lo has utilizado por ahora para atributos y variables (por ejemplo para declarar atributos constantes, que una vez que toman un valor ya no pueden ser modificados). Pero este modificador también puede ser utilizado con clases y con métodos (con un comportamiento que no es exactamente igual, aunque puede encontrarse cierta analogía: **no se permite heredar o no se permite redefinir**).

Una clase declarada como `final` no puede ser heredada, es decir, no puede tener clases derivadas. La jerarquía de clases a la que pertenece acaba en ella (no tendrá clases hijas):

```
1 [modificador_acceso] final class nombreClase [herencia] [interfaces]
```

Un método también puede ser declarado como `final`, en tal caso, ese método no podrá ser redefinido en una clase derivada:

```
1 [modificador_acceso] final <tipo> <nombreMetodo> ([parámetros]) [excepciones]
```

Si intentas redefinir un método `final` en una subclase se producirá un error de compilación.

Distintos contextos en los que puede aparecer el modificador `final`:

Lugar	Función
Como modificador de clase.	La clase no puede tener subclases.
Como modificador de atributo.	El atributo no podrá ser modificado una vez que tome un valor. Sirve para definir constantes.
Como modificador al declarar un método	El método no podrá ser redefinido en una clase derivada.
Como modificador al declarar una variable referencia.	Una vez que la variable tome un valor referencia (un objeto), no se podrá cambiar. La variable siempre apuntará al mismo objeto, lo cual no quiere decir que ese objeto no pueda ser modificado internamente a través de sus métodos. Pero la variable no podrá apuntar a otro objeto diferente.
Como modificador en un parámetro de un método	El valor del parámetro (ya sea un tipo primitivo o una referencia) no podrá modificarse dentro del código del método.

Veamos un ejemplo de cada posibilidad: 1. Modificador de una clase.

```
1 public final class ClaseSinDescendencia { // Clase "no heredable"
2 ...
3 }
```

1. Modificador de un atributo.

```
1 public class ClaseEjemplo {
2     // Valor constante conocido en tiempo de compilación
3     final double PI= 3.14159265;
4
5     // Valor constante conocido solamente en tiempo de ejecución
6     final int SEMILLA= (int) Math.random()*10+1;
7     ...
8 }
```

1. Modificador de un método.

```
1 public final metodoNoRedefinible (int parametro1) { // Método "no redefinible"
2 ...
3 }
```

1. Modificador en una variable referencia.

```
1 // Referencia constante: siempre se apuntará al mismo objeto Alumno recién creado, aunque este objeto pueda sufrir modificaciones.
2 final Alumno PRIMER_ALUMNO= new Alumno ("Pepe", "Torres", 9.55); // Ref. constante
3
4 // Si la variable no es una referencia (tipo primitivo), sería una constante más (como un atributo constante).
5 final int NUMERO_DIEZ= 10; // Valor constante (dentro del ámbito de vida de la variable)
```

1. Modificador en un parámetro de un método.

```
1 void metodoConParametrosFijos (final int par1, final int par2) {
2     // Los parámetros "par1" y "par2" no podrán sufrir modificaciones aquí dentro
3     ...
4 }
```

27.5. Interfaces

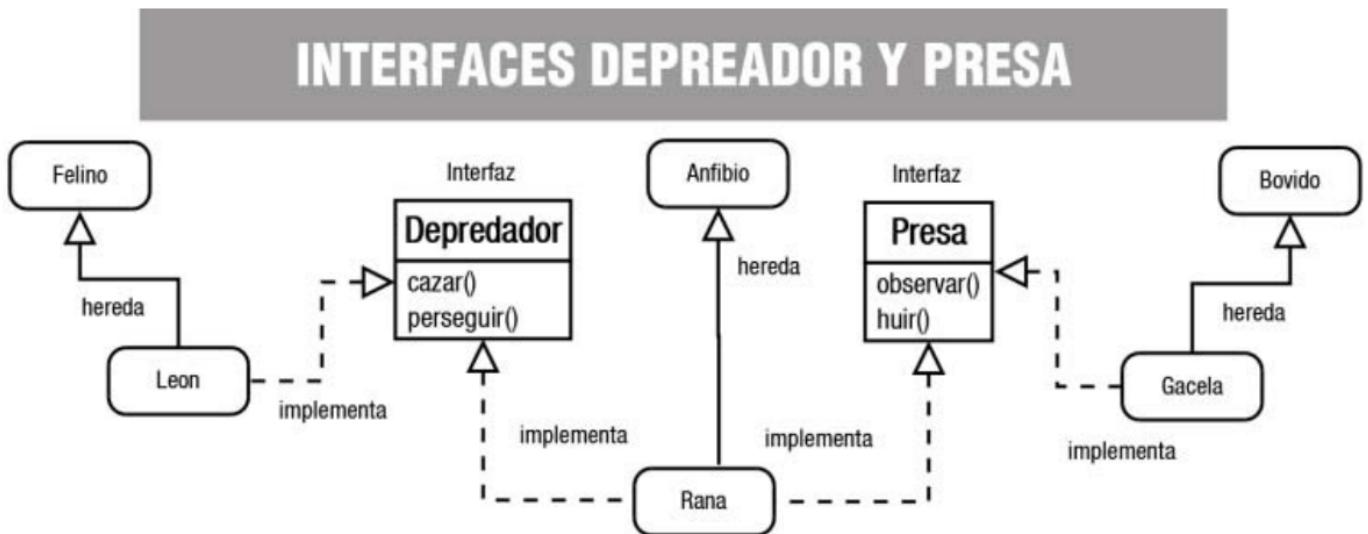
Has visto cómo la herencia permite definir especializaciones (o extensiones) de una clase base que ya existe sin tener que volver a repetir de todo el código de ésta. Este mecanismo da la oportunidad de que la nueva clase especializada (o extendida) disponga de toda la interfaz que tiene su clase base.

También has estudiado cómo los métodos abstractos permiten establecer una interfaz para marcar las líneas generales de un comportamiento común de superclase que deberían compartir de todas las subclases.

Si llevamos al límite esta idea de interfaz, podrías llegar a tener una clase abstracta donde todos sus métodos fueran abstractos. De este modo estarías dando únicamente el marco de comportamiento, sin ningún método implementado, de las posibles subclases que heredarán de esa clase abstracta. La idea de una interfaz (o interface) es precisamente ésa: disponer de un mecanismo que permita especificar cuál debe ser el comportamiento que deben tener todos los objetos que formen parte de una determinada clasificación (no necesariamente jerárquica).

Una interfaz consiste principalmente en una lista de declaraciones de métodos sin implementar, que caracterizan un determinado comportamiento. Si se desea que una clase tenga ese comportamiento, tendrá que implementar esos métodos establecidos en la interfaz. En este caso no se trata de una relación de herencia (la clase A es una especialización de la clase B, o la subclase A es del tipo de la superclase B), sino más bien una relación "de implementación de comportamientos" (la clase A implementa los métodos establecidos en la interfaz B, o los comportamientos indicados por B son llevados a cabo por A; pero no que A sea de clase B).

Imagina que estás diseñando una aplicación que trabaja con clases que representan distintos tipos de animales. Algunas de las acciones que quieras que lleven a cabo están relacionadas con el hecho de que algunos animales sean depredadores (por ejemplo: observar una presa, perseguirla, comérsela, etc.) o sean presas (observar, huir, esconderse, etc.). Si creas la clase `León`, esta clase podría implementar una interfaz `Depredador`, mientras que otras clases como `Gacela` implementarían las acciones de la interfaz `Presa`. Por otro lado, podrías tener también el caso de la clase `Rana`, que implementaría las acciones de la interfaz `Depredador` (pues es cazador de pequeños insectos), pero también la de `Presa` (pues puede ser cazado y necesita las acciones necesarias para protegerse).



27.5.1. Concepto de interfaz.

Una interfaz en Java consiste esencialmente en una lista de declaraciones de métodos sin implementar, junto con un conjunto de constantes.

Estos métodos sin implementar indican un comportamiento, un tipo de conducta, aunque no especifican cómo será ese comportamiento (implementación), pues eso dependerá de las características específicas de cada clase que decida implementar esa interfaz. Podría decirse que una interfaz se encarga de establecer qué comportamientos hay que tener (qué métodos), pero no dice nada de cómo deben llevarse a cabo esos comportamientos (implementación). Se indica sólo la forma, no la implementación.

En cierto modo podrías imaginar el concepto de interfaz como un guión que dice: "éste es el protocolo de comunicación que deben presentar todas las clases que implementen esta interfaz". Se proporciona una lista de métodos públicos y, si quieras dotar a tu clase de esa interfaz, tendrás que definir todos y cada uno de esos métodos públicos.

En conclusión: una interfaz se encarga de establecer unas líneas generales sobre los comportamientos (métodos) que deberían tener los objetos de toda clase que implemente esa interfaz, es decir, que no indican lo que el objeto es (de eso se encarga la clase y sus superclases), sino acciones (capacidades) que el objeto debería ser capaz de realizar. Es por esto que el nombre de muchas interfaces en Java termina con sufijos del tipo "-able", "-or", "-ente" y cosas del estilo, que significan algo así como capacidad o habilidad para hacer o ser receptores de algo (configurable, serializable, modificable, clonable, ejecutable, administrador, servidor, buscador, etc.), dando así la idea de que se tiene la capacidad de llevar a cabo el conjunto de acciones especificadas en la interfaz.

Imagínate por ejemplo la clase `Coche`, subclase de `Vehículo`. Los coches son vehículos a motor, lo cual implica una serie de acciones como, por ejemplo, arrancar el motor o detener el motor. Esta acción no la puedes heredar de `Vehículo`, pues no todos los vehículos tienen porqué ser a motor (piensa por ejemplo en una clase `Bicicleta`), y no puedes heredar de otra clase pues ya heredas de `Vehículo`. Una solución podría ser crear una interfaz `Arrancable`, que proporcione los métodos típicos de un objeto a motor (no necesariamente vehículos). De este modo la clase `Coche` sigue siendo subclase de `Vehículo`, pero también implementaría los comportamientos de la interfaz `Arrancable`, los cuales podrían ser también implementados por otras clases, hereden o no de `Vehículo` (por ejemplo una clase `Motocicleta` o bien una clase `Motosierra`). La clase `Coche` implementará su método arrancar de una manera, la clase `Motocicleta` lo hará de otra (aunque bastante parecida) y la clase `Motosierra` de otra forma probablemente muy diferente, pero todos tendrán su propia versión del método arrancar como parte de la interfaz `Arrancable`.

Según esta concepción, podrías hacerte la siguiente pregunta: ¿podrá una clase implementar varias interfaces? La respuesta en este caso sí es afirmativa.

Importante

Una clase puede adoptar distintos modelos de comportamiento establecidos en diferentes interfaces. **Es decir una clase puede implementar varias interfaces.**

27.5.1.1. ¿CLASE ABSTRACTA O INTERFAZ?

Observando el concepto de interfaz que se acaba de proponer, podría caerse en la tentación de pensar que es prácticamente lo mismo que una clase abstracta en la que todos sus métodos sean abstractos.

Es cierto que en ese sentido existe un gran parecido formal entre una clase abstracta y una interfaz, pudiéndose en ocasiones utilizar indistintamente una u otra para obtener un mismo fin. Pero, a pesar de ese gran parecido, existen algunas diferencias, no sólo formales, sino también conceptuales, muy importantes:

- Una clase no puede heredar de varias clases, aunque sean abstractas (herencia múltiple). Sin embargo sí puede implementar una o varias interfaces y además seguir heredando de una clase.
- Una interfaz no puede definir métodos (no implementa su contenido), tan solo los declara o enumera.
- Una interfaz puede hacer que dos clases tengan un mismo comportamiento independientemente de sus ubicaciones en una determinada jerarquía de clases (no tienen que heredar las dos de una misma superclase, pues no siempre es posible según la naturaleza y propiedades de cada clase).
- Una interfaz permite establecer un comportamiento de clase sin apenas dar detalles, pues esos detalles aún no son conocidos (dependerán del modo en que cada clase decida implementar la interfaz).
- Las interfaces tienen su propia jerarquía, diferente e independiente de la jerarquía de clases.

De todo esto puede deducirse que una clase abstracta proporciona una interfaz disponible sólo a través de la herencia. Sólo quien herede de esa clase abstracta dispondrá de esa interfaz. Si una clase no pertenece a esa misma jerarquía (no hereda de ella) no podrá tener esa interfaz. Eso significa que para poder disponer de la interfaz podrías:

1. Volver a escribirla para esa jerarquía de clases. Lo cual no parece una buena solución.
2. Hacer que la clase herede de la superclase que proporciona la interfaz que te interesa, sacándola de su jerarquía original y convirtiéndola en clase derivada de algo de lo que conceptualmente no debería ser una subclase. Es decir, estarías forzando una relación "es un" cuando en realidad lo más probable es que esa relación no exista. Tampoco parece la mejor forma de resolver el problema.

Sin embargo, una interfaz sí puede ser implementada por cualquier clase, permitiendo que clases que no tengan ninguna relación entre sí (pertenezcan a distintas jerarquías) puedan compartir un determinado comportamiento (una interfaz) sin tener que forzar una relación de herencia que no existe entre ellas.

A partir de ahora podemos hablar de otra posible relación entre clases: la de compartir un determinado comportamiento (interfaz). Dos clases podrían tener en común un determinado conjunto de comportamientos sin que necesariamente exista una relación jerárquica entre ellas. Tan solo cuando haya realmente una relación de tipo "es un" se producirá herencia.

Interfaz o clase abstracta?

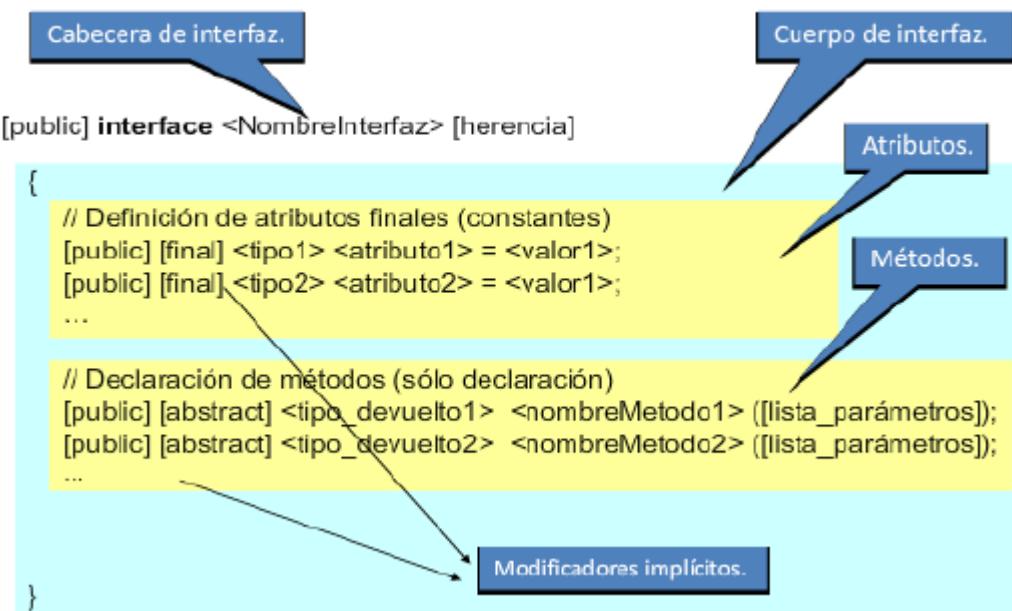
Si sólo vas a proporcionar una lista de métodos abstractos (interfaz), sin definiciones de métodos ni atributos de objeto, suele ser recomendable definir una interfaz antes que clase abstracta. Es más, cuando vayas a definir una supuesta clase base, puedes comenzar declarándola como interfaz y sólo cuando veas que necesitas definir métodos o variables miembro, puedes entonces convertirla en clase abstracta (no instanciable) o incluso en una clase instanciable.

27.5.2. Definición de interfaces.

La declaración de una interfaz en Java es similar a la declaración de una clase, aunque con algunas variaciones:

- Se utiliza la palabra reservada `interface` en lugar de `class`.
- Puede utilizarse el modificador `public`. Si incluye este modificador la interfaz debe tener el mismo nombre que el archivo `.java` en el que se encuentra (exactamente igual que sucedía con las clases). Si no se indica el modificador `public`, el acceso será por omisión o "de paquete" (como sucedía con las clases).
- Todos los miembros de la interfaz (atributos y métodos) son `public` de manera implícita. No es necesario indicar el modificador `public`, aunque puede hacerse.
- Todos los atributos son de tipo `final` y `public` (tampoco es necesario especificarlo), es decir, constantes y públicos. Hay que darles un valor inicial.
- Todos los métodos son abstractos también de manera implícita (tampoco hay que indicarlo). No tienen cuerpo, tan solo la cabecera.

Interfaces en Java



Como puedes observar, una interfaz consiste esencialmente en una lista de atributos finales (constantes) y métodos abstractos (sin implementar). Su sintaxis quedaría entonces:

```

1  [public] interface <NombreInterfaz> {
2      [public] [final] <tipo1> <atributo1>=<valor1>;
3      [public] [final] <tipo2> <atributo2>=<valor2>;
4      ...
5      [public] [abstract] <tipo_devuelto1> <nombreMetodo1> ([lista_parámetros]);
6      [public] [abstract] <tipo_devuelto2> <nombreMetodo2> ([lista_parámetros]);
7      ...
8  }
  
```

Si te fijas, la declaración de los métodos termina en punto y coma, pues no tienen cuerpo, al igual que sucede con los métodos abstractos de las clases abstractas. El ejemplo de la interfaz `Depredador` que hemos visto antes podría quedar entonces así:

```

1  public interface Depredador {
2      void perseguir (Animal presa);
3      void cazar (Animal presa);
4      ...
5  }

```

Serán las clases que implementen esta interfaz (`León`, `Leopardo`, `Cocodrilo`, `Rana`, `Lagarto`, `Hombre`, etc.) las que definan cada uno de los métodos por dentro.

Revisa con cuidado el [Ejemplo 5.2](#)

27.5.3. Implementación de interfaces.

Como ya has visto, todas las clases que implementan una determinada interfaz están obligadas a proporcionar una definición (implementación) de los métodos de esa interfaz, adoptando el modelo de comportamiento propuesto por ésta.

Dada una interfaz, cualquier clase puede especificar dicha interfaz mediante el mecanismo denominado implementación de interfaces. Para ello se utiliza la palabra reservada `implements`:

```

1  class NombreClase implements NombreInterfaz {

```

De esta manera, la clase está diciendo algo así como "la interfaz indica los métodos que debo implementar, pero voy a ser yo (la clase) quien los implemente".

Es posible indicar varios nombres de interfaces separándolos por comas:

```

1  class NombreClase implements NombreInterfaz1, NombreInterfaz2, ...

```

Cuando una clase implementa una interfaz, tiene que redefinir sus métodos nuevamente con acceso público. Con otro tipo de acceso se producirá un error de compilación. Es decir, que del mismo modo que no se podían restringir permisos de acceso en la herencia de clases, tampoco se puede hacer en la implementación de interfaces.

Una vez implementada una interfaz en una clase, los métodos de esa interfaz tienen exactamente el mismo tratamiento que cualquier otro método, sin ninguna diferencia, pudiendo ser invocados, heredados, redefinidos, etc.

En el ejemplo de los depredadores, al definir la clase `León`, habría que indicar que implementa la interfaz `Depredador`:

```

1  class Leon implements Depredador {

```

En realidad la definición completa de la clase `Leon` debería ser:

```

1  class Leon extends Felino implements Depredador {

```

Orden importa

El orden de `extends` e `implements` es importante, primero se define la herencia y a continuación la interfaces que implementa.

Y en su interior habría que implementar aquellos métodos que contenga la interfaz:

```

1  void perseguir (Animal presa) {
2      // Implementación del método localizar para un león
3      ...
4  }

```

En el caso de clases que pudieran ser a la vez `Depredador` y `Presa`, tendrían que implementar ambas interfaces, como podría suceder con la clase `Rana`:

```

1  class Rana implements Depredador, Presa {

```

Que de manera completa quedaría:

```

1  class Rana extends Anfibio implements Depredador, Presa {

```

Y en su interior habría que implementar aquellos métodos que contengan ambas interfaces, tanto las de Depredador (localizar, cazar, etc.) como las de Presa (observar, huir, etc.).

Revisa con cuidado el [Ejemplo 5.3](#)

27.5.3.1. UN EJEMPLO DE IMPLEMENTACIÓN DE INTERFACES: LA INTERFAZ SERIES.

En la forma tradicional de una interfaz, los métodos se declaran utilizando solo su tipo de devolución y firma. Son, esencialmente, métodos abstractos. Por lo tanto, cada clase que incluye dicha interfaz debe implementar todos sus métodos.

Recuerda

En una interfaz, los métodos son implícitamente públicos.

Recuerda

Las variables declaradas en una interfaz no son variables de instancia. En cambio, son implícitamente `public`, `final`, y `static`, y deben inicializarse. Por lo tanto, son esencialmente **constantes**.

Aquí hay un ejemplo de una definición de interfaz. Especifica la interfaz a una clase que genera una serie de números.

```
1 public interface Series {
2     int getSiguiente(); //Retorna el siguiente número de la serie
3     void reiniciar(); //Reinicia
4     void setComenzar(int x); //Establece un valor inicial
5 }
```

Esta interfaz se declara pública para que pueda ser implementada por código en cualquier paquete.

Los métodos que implementan una interfaz deben declararse públicos. Además, el tipo del método de implementación debe coincidir exactamente con el tipo especificado en la definición de la interfaz.

Aquí hay un ejemplo que implementa la interfaz de `Series` mostrada anteriormente. Crea una clase llamada `DeDos`, que genera una serie de números, cada uno mayor que el anterior.

```
1 class DeDos implements Series {
2     int iniciar;
3     int valor;
4
5     DeDos(){
6         iniciar=0;
7         valor=0;
8     }
9
10    public int getSiguiente() {
11        valor+=2;
12        return valor;
13    }
14
15    public void reiniciar() {
16        valor=iniciar;
17    }
18
19    public void setComenzar(int x) {
20        iniciar=x;
21        valor=x;
22    }
23 }
```

Observe que los métodos `getSiguiente()`, `reiniciar()` y `setComenzar()` se declaran utilizando el especificador de acceso público (`public`). Esto es necesario. Siempre que implemente un método definido por una interfaz, debe implementarse como público porque todos los miembros de una interfaz son implícitamente públicos.

Aquí hay una clase que demuestra `DeDos`:

```

1 class SeriesDemo {
2     public static void main(String[] args) {
3         DeDos ob=new DeDos();
4         for (int i=0;i<5;i++){
5             System.out.println("Siguiente valor es: "+ob.getSiguiente());
6         }
7         System.out.println("\nReiniciando");
8         ob.reiniciar();
9         for (int i=0;i<5;i++){
10            System.out.println("Siguiente valor es: "+ob.getSiguiente());
11        }
12        System.out.println("\nIniciando en 100");
13        ob.setComenzar(100);
14        for (int i=0;i<5;i++){
15            System.out.println("Siguiente valor es: "+ob.getSiguiente());
16        }
17    }
18 }
```

Salida:

```

1 Siguiente valor es: 2
2 Siguiente valor es: 4
3 Siguiente valor es: 6
4 Siguiente valor es: 8
5 Siguiente valor es: 10
6 Reiniciando
7 Siguiente valor es: 2
8 Siguiente valor es: 4
9 Siguiente valor es: 6
10 Siguiente valor es: 8
11 Siguiente valor es: 10
12 Iniciando en 100
13 Siguiente valor es: 102
14 Siguiente valor es: 104
15 Siguiente valor es: 106
16 Siguiente valor es: 108
17 Siguiente valor es: 110
```

Está permitido y es común para las clases que implementan interfaces definir miembros adicionales propios. Por ejemplo, la siguiente versión de `DeDos` agrega el método `getAnterior()`, que devuelve el valor anterior:

```

1 class DeDos implements Series {
2     int iniciar;
3     int valor;
4     int anterior;
5     DeDos(){
6         iniciar=0;
7         valor=0;
8     }
9     public int getSiguiente() {
10        anterior=valor;
11        valor+=2;
12        return valor;
13    }
14    public void reiniciar() {
15        valor=iniciar;
16        anterior=valor-2;
17    }
18    public void setComenzar(int x) {
19        iniciar=x;
20        valor=x;
21        anterior=x-2;
22    }
23    //Añadiendo un método que no está definido en Series
24    int getAnterior(){
25        return anterior;
26    }
27 }
```

Observe que la adición de `getAnterior()` requirió un cambio en las implementaciones de los métodos definidos por `Series`. Sin embargo, dado que la interfaz con esos métodos permanece igual, el cambio es continuo y no rompe el código preexistente. Esta es una de las ventajas de las interfaces.

Como se explicó, cualquier cantidad de clases puede implementar una interfaz. Por ejemplo, aquí hay una clase llamada `DeTres` que genera una serie que consta de múltiplos de tres:

```

1  public class DeTres implements Series{
2      int iniciar;
3      int valor;
4      DeTres(){
5          iniciar=0;
6          valor=0;
7      }
8      public int getSiguiente() {
9          valor+=3;
10         return valor;
11     }
12     public void reiniciar() {
13         valor=iniciar;
14     }
15     public void setComenzar(int x) {
16         iniciar=x;
17         valor=x;
18     }
19 }
```

27.5.4. Simulación de la herencia múltiple mediante el uso de interfaces.

Una interfaz no tiene espacio de almacenamiento asociado (no se van a declarar objetos de un tipo de interfaz), es decir, no tiene implementación.

En algunas ocasiones es posible que interese representar la situación de que "una clase X es de tipo A, de tipo B, y de tipo C", siendo A, B, C clases disjuntas (no heredan unas de otras). Hemos visto que sería un caso de herencia múltiple que Java no permite.

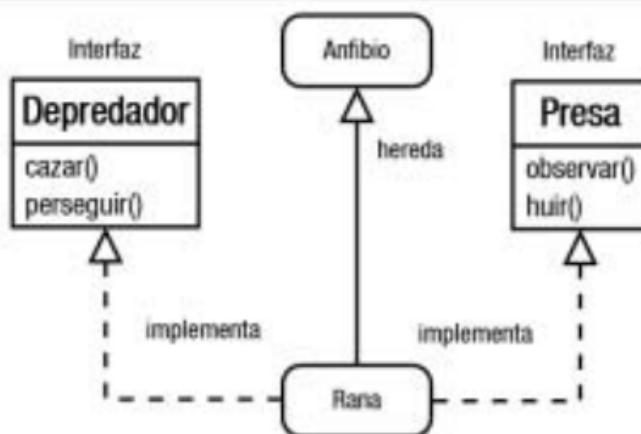
Para poder simular algo así, podrías definir tres interfaces A, B, C que indiquen los comportamientos (métodos) que se deberían tener según se pertenezca a una supuesta clase A, B, o C, pero sin implementar ningún método concreto ni atributos de objeto (sólo interfaz).

De esta manera la clase X podría a la vez:

1. Implementar las interfaces A, B, C, que la dotarían de los comportamientos que deseaba heredar de las clases A, B, C.
2. Heredar de otra clase Y, que le proporcionaría determinadas características dentro de su taxonomía o jerarquía de objeto (atributos, métodos implementados y métodos abstractos).

En el ejemplo que hemos visto de las interfaces `Depredador` y `Presa`, tendrías un ejemplo de esto: la clase `Rana`, que es subclase de `Anfibio`, implementa una serie de comportamientos propios de un `Depredador` y, a la vez, otros más propios de una `Presa`. Esos comportamientos (métodos) no forman parte de la superclase `Anfibio`, sino de las interfaces. Si se decide que la clase `Rana` debe de llevar a cabo algunos otros comportamientos adicionales, podrían añadirse a una nueva interfaz y la clase `Rana` implementaría una tercera interfaz.

IMPLEMENTACIÓN DE LAS INTERFACES DEPREDADOR Y PRESA



De este modo, con el mecanismo "**una herencia pero varias interfaces**", podrían conseguirse resultados similares a los obtenidos con la herencia múltiple.

Ahora bien, del mismo modo que sucedía con la herencia múltiple, puede darse el problema de la colisión de nombres al implementar dos interfaces que tengan un método con el mismo identificador. En tal caso puede suceder lo siguiente:

- Si los dos métodos tienen diferentes parámetros no habrá problema aunque tengan el mismo nombre pues se realiza una sobrecarga de métodos.
- Si los dos métodos tienen un valor de retorno de un tipo diferente, se producirá un error de compilación (al igual que sucede en la sobrecarga cuando la única diferencia entre dos métodos es ésa).

Si los dos métodos son exactamente iguales en identificador, parámetros y tipo devuelto, entonces solamente se podrá implementar uno de los dos métodos. En realidad se trata de un solo método pues ambos tienen la misma interfaz (mismo identificador, mismos parámetros y mismo tipo devuelto).

Cuidado!

La utilización de nombres idénticos en diferentes interfaces que pueden ser implementadas a la vez por una misma clase puede causar, además del problema de la colisión de nombres, dificultades de legibilidad en el código, pudiendo dar lugar a confusiones. Si es posible intenta evitar que se produzcan este tipo de situaciones.

27.5.5. Herencia de interfaces.

Las interfaces, al igual que las clases, también permiten la herencia. Para indicar que una interfaz hereda de otra se indica nuevamente con la palabra reservada `extends`. **Pero en este caso sí se permite la herencia múltiple de interfaces.** Si se hereda de más de una interfaz se indica con la lista de interfaces separadas por comas.

Por ejemplo, dadas las interfaces `InterfazUno` e `InterfazDos`:

```
1 public interface InterfazUno {
2     // Métodos y constantes de la interfaz Uno
3 }
4
5 public interface InterfazDos {
6     // Métodos y constantes de la interfaz Dos
7 }
```

Podría definirse una nueva interfaz que heredara de ambas:

```
1 public interface InterfazCompleja extends InterfazUno, InterfazDos {
2     // Métodos y constantes de la interfaz compleja
3 }
```

Revisa con cuidado el [Ejemplo 5.5](#) y también el [Ejemplo 5.6](#)

27.5.6. Funciones Lambda

Tal y como vimos en la unidad anterior, la implementación de los métodos de interfaces es muy susceptible de serlo a través de funciones lambda.

Imaginemos una clase `Persona`:

```
1 class Persona{
2     private String nombre;
3     private int edad;
4     ...
5 }
```

Y un `ArrayList` `personas` formada por objetos de tipo `Persona`:

```
1 ...
2 ArrayList<Persona> personas = new ArrayList<>();
3 personas.add(new Persona("Nacho", 52));
4 personas.add(new Persona("David", 47));
5 personas.add(new Persona("Pepe", 42));
6 personas.add(new Persona("Maria", 22));
7 personas.add(new Persona("Marta", 4));
8 ...
```

Ahora queremos ordenar el `ArrayList` de `personas` de mayor a menor edad usando... Implementación "tradicional" java:
`Comparator` o `Comparable`

```

1 ...
2 class ComparadorPersona implements Comparator <Persona>{
3     @Override
4     public int compare(Persona p1, Persona p2){
5         return p2.getEdad() - p1.getEdad();
6     }
7 }
8 ...

```

```

1 ...
2 personas.sort(new ComparadorPersona());
3 for (int i = 0; i < personas.size(); i++){
4     System.out.println(personas.get(i));
5 }
6 ...

```

Sin embargo, implementado con funciones Lambda sería...

```

1 ...
2 personas.sort((p1, p2) -> p2.getEdad() - p1.getEdad());
3 for (int i = 0; i < personas.size(); i++){
4     System.out.println(personas.get(i));
5 }
6 ...

```

27.6. Polimorfismo

El polimorfismo es otro de los grandes pilares sobre los que se sustenta la Programación Orientada a Objetos (junto con la encapsulación y la herencia). Se trata nuevamente de otra forma más de establecer diferencias entre interfaz e implementación, es decir, entre el qué y el cómo.

La **encapsulación** te ha permitido agrupar características (atributos) y comportamientos (métodos) dentro de una misma unidad (clase), pudiendo darles un mayor o menor componente de visibilidad, y permitiendo separar al máximo posible la interfaz de la implementación.

Por otro lado la **herencia** te ha proporcionado la posibilidad de tratar a los objetos como pertenecientes a una jerarquía de clases. Esta capacidad va a ser fundamental a la hora de poder manipular muchos posibles objetos de clases diferentes como si fueran de la misma clase (polimorfismo).

El **polimorfismo** te va a permitir mejorar la organización y la legibilidad del código así como la posibilidad de desarrollar aplicaciones que sean más fáciles de ampliar a la hora de incorporar nuevas funcionalidades. Si la implementación y la utilización de las clases es lo suficientemente genérica y extensible será más sencillo poder volver a este código para incluir nuevos requerimientos.

27.6.1. Concepto de polimorfismo.

El polimorfismo consiste en la capacidad de poder utilizar una referencia a un objeto de una determinada clase como si fuera de otra clase (en concreto una subclase). Es una manera de decir que una clase podría tener varias (poli) formas (morfismo).

Un método "polimórfico" ofrece la posibilidad de ser distinguido (saber a qué clase pertenece) en tiempo de ejecución en lugar de en tiempo de compilación. Para poder hacer algo así es necesario utilizar métodos que pertenecen a una superclase y que en cada subclase se implementan de una forma en particular. En tiempo de compilación se invocará al método sin saber exactamente si será el de una subclase u otra (pues se está invocando al de la superclase). Sólo en tiempo de ejecución (una vez instanciada una u otra subclase) se conocerá realmente qué método (de qué subclase) es el que finalmente va a ser invocado.

Esta forma de trabajar te va a permitir hasta cierto punto "desentenderte" del tipo de objeto específico (subclase) para centrarte en el tipo de objeto genérico (superclase). De este modo podrás manipular objetos hasta cierto punto "desconocidos" en tiempo de compilación y que sólo durante la ejecución del programa se sabrá exactamente de qué tipo de objeto (subclase) se trata.

Polimorfismo

El polimorfismo ofrece la posibilidad de que toda referencia a un objeto de una superclase pueda tomar la forma de una referencia a un objeto de una de sus subclases. Esto te va a permitir escribir programas que procesen objetos de clases que formen parte de la misma jerarquía como si todos fueran objetos de sus superclases.

Más polimorfismo

El polimorfismo puede llevarse a cabo tanto con superclases (abstractas o no) como con interfaces.

Dada una superclase X, con un método m, y dos subclases A y B, que redefinen ese método m, podrías declarar un objeto O de tipo X que durante la ejecución podrá ser de tipo A o de tipo B (algo desconocido en tiempo de compilación). Esto significa que al invocarse el método m de X (superclase), se estará en realidad invocando al método m de A o de B (alguna de sus subclases). Por ejemplo:

```

1 // Declaración de una referencia a un objeto de tipo X
2 ClaseX obj; // Objeto de tipo X (superclase)
3 ...
4
5 // Zona del programa donde se instancia un objeto de tipo A (subclase) y se le asigna a la referencia obj.
6 // La variable obj adquiere la forma de la subclase A.
7 obj = new ClaseA();
8 ...
9
10 // Otra zona del programa.
11 // Aquí se instancia un objeto de tipo B (subclase) y se le asigna a la referencia obj.
12 // La variable obj adquiere la forma de la subclase B.
13 obj = new ClaseB();
14 ...
15
16 // Zona donde se utiliza el método m sin saber realmente qué subclase se está utilizando.
17 // (Sólo se sabrá durante la ejecución del programa)
18
19 obj.m()
20 // Llamada al método m (sin saber si será el método m de A o de B).
21 ...

```

Imagina que estás trabajando con las clases `Alumno` y `Profesor` y que en determinada zona del código podrías tener objetos, tanto de un tipo como de otro, pero eso sólo se sabrá según vaya discurriendo la ejecución del programa. En algunos casos, es posible que un determinado objeto pudiera ser de la clase `Alumno` y en otros de la clase `Profesor`, pero en cualquier caso serán objetos de la clase `Persona`. Eso significa que la llamada a un método de la clase `Persona` (por ejemplo `devolverContenidoString`) en realidad será en unos casos a un método (con el mismo nombre) de la clase `Alumno` y, en otros, a un método (con el mismo nombre también) de la clase `Profesor`. Esto será posible hacerlo gracias a la ligadura dinámica.

27.6.2. Ligadura dinámica.

La conexión que tiene lugar durante una llamada a un método suele ser llamada ligadura (conexión o vinculación que tiene lugar durante una llamada a un método para saber qué código debe ser ejecutado. Puede ser estática o dinámica, vinculación o enlace (en inglés binding). Si esta vinculación se lleva a cabo durante el proceso de compilación, se le suele llamar ligadura estática (la vinculación que se produce en la llamada a un método con la clase a la que pertenece ese método se realiza en tiempo de compilación. Es decir, que antes de generar el código ejecutable se conoce exactamente el método (a qué clase pertenece) que será llamado. También conocido como vinculación temprana). En los lenguajes tradicionales, no orientados a objetos, ésta es la única forma de poder resolver la ligadura (en tiempo de compilación). Sin embargo, en los lenguajes orientados a objetos existe otra posibilidad: la ligadura dinámica (la vinculación que se produce en la llamada a un método con la clase a la que pertenece ese método se realiza en tiempo de ejecución. Es decir, que al generar el código ejecutable no se conoce exactamente el método (a qué clase pertenece) que será llamado. Sólo se sabrá cuando el programa esté en ejecución. También conocida como vinculación tardía, enlace tardío o late binding).

La ligadura dinámica hace posible que sea el tipo de objeto instanciado (obtenido mediante el constructor finalmente utilizado para crear el objeto) y no el tipo de la referencia (el tipo indicado en la declaración de la variable que apuntará al objeto) lo que determine qué versión del método va a ser invocada. El tipo de objeto al que apunta la variable de tipo referencia sólo podrá ser conocido durante la ejecución del programa y por eso el polimorfismo necesita la ligadura dinámica.

En el ejemplo anterior de la clase X y sus subclases A y B, la llamada al método m sólo puede resolverse mediante ligadura dinámica, pues es imposible saber en tiempo de compilación si el método m que debe ser invocado será el definido en la subclase A o el definido en la subclase B:

```

1 //Llamada al método m (sin saber si será el método m de A o de B).
2 obj.m() // Esta llamada será resuelta en tiempo de ejecución (ligadura dinámica)

```

Revisa con cuidado el [Ejemplo 6.2](#)

27.6.3. Limitaciones de la ligadura dinámica.

Como has podido comprobar, el polimorfismo se basa en la utilización de referencias de un tipo más "amplio" (superclases) que los objetos a los que luego realmente van a apuntar (subclases). Ahora bien, existe una importante restricción en el uso de esta capacidad, pues el tipo de referencia limita cuáles son los métodos que se pueden utilizar y los atributos a los que se pueden acceder.

Importante

No se puede acceder a los miembros específicos de una subclase a través de una referencia a una superclase. Sólo se pueden utilizar los miembros declarados en la superclase, aunque la definición que finalmente se utilice en su ejecución sea la de la subclase.

Veamos un ejemplo: si dispones de una clase `Profesor` que es subclase de `Persona` y declaras una variable como referencia un objeto de tipo `Persona`. Aunque más tarde esa variable haga referencia a un objeto de tipo `Profesor` (subclase), los miembros a los que podrás acceder sin que el compilador produzca un error serán los miembros de `Profesor` que hayan sido heredados de `Persona` (superclase). De este modo, se garantiza que los métodos que se intenten llamar van a existir cualquiera que sea la subclase de `Persona` a la que se apunte desde esa referencia.

En el ejemplo de las clases `Persona`, `Profesor` y `Alumno`, el polimorfismo nos permitiría declarar variables de tipo `Persona` y más tarde hacer con ellas referencia a objetos de tipo `Profesor` o `Alumno`, pero no deberíamos intentar acceder con esa variable a métodos que sean específicos de la clase `Profesor` o de la clase `Alumno`, tan solo a métodos que sabemos que van a existir seguro en ambos tipos de objetos (métodos de la superclase `Persona`).

Revisa con cuidado el [Ejemplo 6.3](#)

27.6.4. Interfaces y polimorfismo.

Es posible también llevar a cabo el polimorfismo mediante el uso de interfaces. Un objeto puede tener una referencia cuyo tipo sea una interfaz, pero para que el compilador te lo permita, la clase cuyo constructor se utilice para crear el objeto deberá implementar esa interfaz (bien por sí misma o bien porque la implemente alguna superclase). Un objeto cuya referencia sea de tipo interfaz sólo puede utilizar aquellos métodos definidos en la interfaz, es decir, que no podrán utilizarse los atributos y métodos específicos de su clase, tan solo los de la interfaz.

Las referencias de tipo interfaz permiten unificar de una manera bastante estricta la forma de utilizarse de objetos que pertenezcan a clases muy diferentes (pero que todas ellas implementan la misma interfaz). De este modo podrías hacer referencia a diferentes objetos que no tienen ninguna relación jerárquica entre sí utilizando la misma variable (referencia a la interfaz). Lo único que los distintos objetos tendrían en común es que implementan la misma interfaz.

Recuerda

En este caso sólo podrás llamar a los métodos de la interfaz y no a los específicos de las clases.

Por ejemplo, si tenías una variable de tipo referencia a la interfaz `Arrancable`, podrías instanciar objetos de tipo `Coche` o `Motosierra` y asignarlos a esa referencia (teniendo en cuenta que ambas clases no tienen una relación de herencia). Sin embargo, tan solo podrás usar en ambos casos los métodos y los atributos de la interfaz `Arrancable` (por ejemplo arrancar) y no los de `Coche` o los de `Motosierra` (sólo los genéricos, nunca los específicos).

En el caso de las clases `Persona`, `Alumno` y `Profesor`, podrías declarar, por ejemplo, variables del tipo `Imprimible`:

```
1 Imprimible obj; // Imprimible es una interfaz y no una clase
```

Con este tipo de referencia podrías luego apuntar a objetos tanto de tipo `Profesor` como de tipo `Alumno`, pues ambos implementan la interfaz `Imprimible`:

```
1 // En algunas circunstancias podría suceder esto:
2 obj= new Alumno (nombre, apellidos, fecha, grupo, nota); // Polimorfismo con interfaces
3 ...
4
5 // En otras circunstancias podría suceder esto:
6 obj= new Profesor (nombre, apellidos, fecha, especialidad, salario); // Polimorfismo con interfaces
7 ...
```

Y más adelante hacer uso de la ligadura dinámica:

```

1 // Llamadas sólo a métodos de la interfaz
2 String contenido;
3 contenido= obj.devolverContenidoString(); // Ligadura dinámica con interfaces

```

27.6.5. Conversión de objetos.

Como ya has visto, en principio no se puede acceder a los miembros específicos de una subclase a través de una referencia a una superclase. Si deseas tener acceso a todos los métodos y atributos específicos del objeto subclase tendrás que realizar una conversión explícita (casting) que convierta la referencia más general (superclase) en la del tipo específico del objeto (subclase).

Para que puedas realizar conversiones entre distintas clases es obligatorio que exista una relación de herencia entre ellas (una debe ser clase derivada de la otra). Se realizará una conversión implícita o automática de subclase a superclase siempre que sea necesario, pues un objeto de tipo subclase siempre contendrá toda la información necesaria para ser considerado un objeto de la superclase.

Ahora bien, la conversión en sentido contrario (de superclase a subclase) debe hacerse de forma explícita y según el caso podría dar lugar a errores por falta de información (atributos) o de métodos. En tales casos se produce una excepción de tipo `ClassCastException`.

Por ejemplo, imagina que tienes una clase `Animal` y una clase `Besugo`, subclase de `Animal`:

```

1 class Animal {
2     public String nombre;
3 }
4 class Besugo extends Animal {
5     public double peso;
6 }

```

A continuación declaras una variable referencia a la clase `Animal` (superclase) pero sin embargo le asignas una referencia a un objeto de la clase `Besugo` (subclase) haciendo uso del polimorfismo:

```

1 Animal obj; // Referencia a objetos de la clase Animal
2 obj= new Besugo(); // Referencia a objetos clase Animal, pero apunta realmente a objeto clase Besugo (polimorfismo)

```

El objeto que acabas de crear como instancia de la clase `Besugo` (subclase de `Animal`) contiene más información que la que la referencia `obj` te permite en principio acceder sin que el compilador genere un error (pues es de clase `Animal`). En concreto los objetos de la clase `Besugo` disponen de `nombre` y `peso`, mientras que los objetos de la clase `Animal` sólo de `nombre`. Para acceder a esa información adicional de la clase especializada (`peso`) tendrás que realizar una conversión explícita (casting):

```

1 // Casting del tipo Animal al tipo Besugo (funcionará bien porque el objeto es realmente del tipo B)
2 System.out.printf ("obj.peso=%f\n", ((Besugo) obj).peso);

```

Sin embargo si se hubiera tratado de una instancia de la clase `Animal` y hubieras intentado acceder al miembro `peso`, se habría producido una excepción de tipo `ClassCastException`:

```

1 Animal obj; // Referencia a objetos de la clase Animal
2 obj= new Animal (); // Referencia a objetos de la clase Animal, y apunta realmente a un objeto de la clase Animal
3
4 // Casting del tipo Animal al tipo Besugo (puede dar problemas porque el objeto es realmente del tipo Animal):
5 // Funciona (la clase Animal tiene nombre)
6 System.out.printf ("obj.nombre=%s\n", ((Besugo) obj).nombre);
7
8 // ¡Error en ejecución! (la clase Animal no tiene peso). Producirá una ClassCastException.
9 System.out.printf ("obj.peso=%f\n", ((Besugo) obj).peso);

```

27.7. Ejemplos UD08

27.7.1. Ejemplo 2.1

Intenta rescribir los siguientes los métodos de la clase `Rectangulo` teniendo en cuenta ahora su nueva estructura de atributos (dos objetos de la clase `Punto`, en lugar de cuatro elementos de tipo `double`):

1. Método `calcularSuperficie`, que calcula y devuelve el área de la superficie encerrada por la figura.
2. Método `calcularPerimetro`, que calcula y devuelve la longitud del perímetro de la figura.

En ambos casos la interfaz no se ve modificada en absoluto (desde fuera su funcionamiento es el mismo), pero internamente deberás tener en cuenta que ya no existen los atributos `x1`, `y1`, `x2`, `y2`, de tipo `double`, sino los atributos `vertice1` y `vertice2` de tipo `Punto`.

Clase Punto :

```

1 package UD08._01_Ejemplo_2_1;
2
3 public class Punto {
4     private double x;
5     private double y;
6
7     public Punto(double x, double y) {
8         this.x = x;
9         this.y = y;
10    }
11
12    public double getX() {
13        return x;
14    }
15
16    public void setX(double x) {
17        this.x = x;
18    }
19
20    public double getY() {
21        return y;
22    }
23
24    public void setY(double y) {
25        this.y = y;
26    }
27 }
```

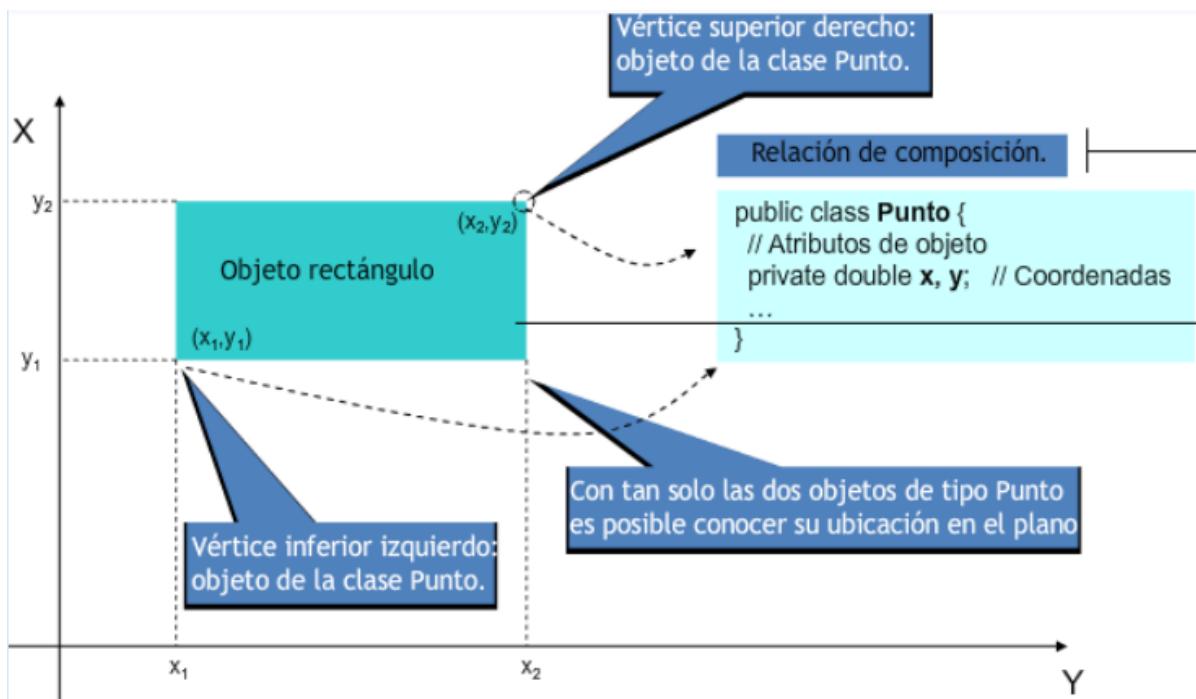
Clase Rectangulo :

```

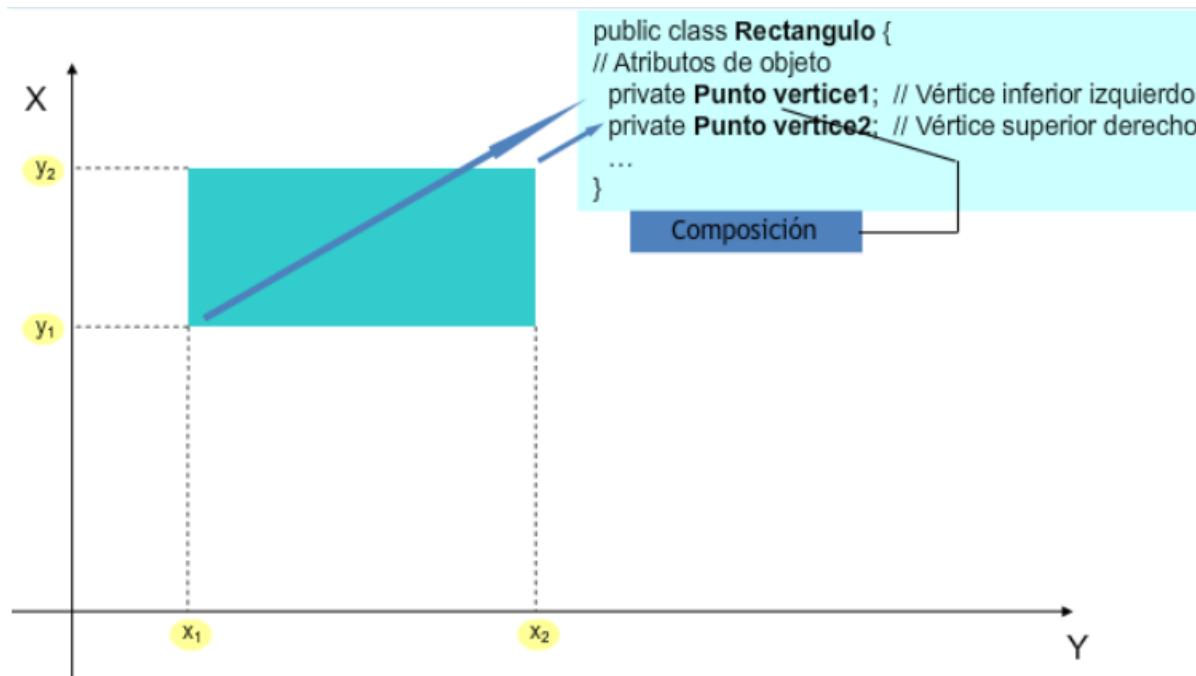
1 package UD08._01_Ejemplo_2_1;
2
3 class Rectangulo {
4
5     private Punto vertice1;
6     private Punto vertice2;
7
8     public double calcularSuperficie() {
9         double area, base, altura; // Variables locales
10        base = vertice2.getX() - vertice1.getX(); // Antes era x2 - x1
11        altura = vertice2.getY() - vertice1.getY(); // Antes era y2 - y1
12        area = base * altura;
13        return area;
14    }
15
16    public double CalcularPerimetro() {
17        double perimetro, base, altura; // Variables locales
18        base = vertice2.getX() - vertice1.getX(); // Antes era x2 - x1
19        altura = vertice2.getY() - vertice1.getY(); // Antes era y2 - y1
20        perimetro = 2 * base + 2 * altura;
21        return perimetro;
22    }
23 }
```

En la siguiente presentación puedes observar detalladamente el proceso completo de elaboración de la clase `Rectangulo` haciendo uso de la clase `Punto` :

1. Objetos de tipo `Rectangulo` compuesto por objetos de tipo `Punto`



1. Atributos de un objeto Rectangulo compuesto por objetos de tipo Punto



1. Clase Rectangulo

A partir de ahora los rectángulos serán representados mediante objetos de esta clase (contenedores de dos objetos Punto).

```
public class Rectangulo {
    // Atributos de objeto
    private Punto vertice1; // Vértice inferior izquierdo
    private Punto vertice2; // Vértice superior derecho
    ...
}
```

1. Método calcularSuperficie

Manipulación de los objetos contenidos a través de los métodos de esos objetos.

```
public double calcularSuperficie () {
    double area, base, altura;

    // Cálculo de la base
    base= vertice2.obtenerX () - vertice1.obtenerX ();
    // Cálculo de la altura
    altura= vertice2.obtenerY () - vertice1.obtenerY ();
    // Cálculo del área
    area= base*altura;

    return area; // Valor de retorno
}
```

```
public class Rectangulo {
    ...
    // Atributos de objeto
    private Punto vertice1; private Punto vertice2;
```

double x
double y

double x
double y

27.7.2. Ejemplo 2.2

Dada la clase `Rectangulo`, escribe sus nuevos métodos `obtenerVertice1` y `obtenerVertice2` para que devuelvan los vértices inferior izquierdo y superior derecho del rectángulo (objetos de tipo `Punto`), teniendo en cuenta su nueva estructura de atributos (dos objetos de la clase `Punto`, en lugar de cuatro elementos de tipo `double`):

Los métodos de obtención de vértices devolverán objetos de la clase `Punto`:

```
1  public Punto obtenerVertice1 (){
2      return vertice1;
3  }
4
5  public Punto obtenerVertice2 (){
6      return vertice2;
7  }
```

Esto funcionaría perfectamente, pero deberías tener cuidado con este tipo de métodos que devuelven directamente una referencia a un objeto atributo que probablemente has definido como privado. Estás de alguna manera haciendo público un atributo que fue declarado como privado.

Para evitar que esto suceda bastaría con crear un nuevo objeto que fuera una copia del atributo que se desea devolver (en este caso un objeto de la clase `Punto`).

Aquí tienes la solución para la nueva clase `Rectangulo`:

```

1 package UD08._02_Ejemplo_2_2;
2
3 class Rectangulo {
4
5     private Punto vertice1;
6     private Punto vertice2;
7
8     public double calcularSuperficie() {
9         double area, base, altura; // Variables locales
10        base = vertice2.getX() - vertice1.getX(); // Antes era x2 - x1
11        altura = vertice2.getY() - vertice1.getY(); // Antes era y2 - y1
12        area = base * altura;
13        return area;
14    }
15
16    public double CalcularPerimetro() {
17        double perimetro, base, altura; // Variables locales
18        base = vertice2.getX() - vertice1.getX(); // Antes era x2 - x1
19        altura = vertice2.getY() - vertice1.getY(); // Antes era y2 - y1
20        perimetro = 2 * base + 2 * altura;
21        return perimetro;
22    }
23
24    /*
25     * Así no!
26     *
27     *public Punto obtenerVertice1mal() {
28     *    return vertice1;
29     *}
23
31     *public Punto obtenerVertice2mal() {
32     *    return vertice2;
33     *}
34     */
35
36     //Mejor de este modo
37     public Punto obtenerVertice1() {
38         // Creación de un nuevo punto extrayendo sus atributos
39         double x, y;
40         Punto p;
41         x = this.vertice1.getX();
42         y = this.vertice1.getY();
43         p = new Punto(x, y);
44         return p;
45     }
46
47     //O mejor así:
48     public Punto obtenerVertice2() {
49         // Utilizando el constructor copia de Punto (si es que está definido)
50         //Punto p;
51         //p = new Punto(this.vertice2); // Uso del constructor copia
52         //return p;
53
54         //o más corto:
55         return new Punto(this.vertice2);
56     }
57
58     public Rectangulo(Punto vertice1, Punto vertice2) {
59         this.vertice1 = vertice1;
60         this.vertice2 = vertice2;
61     }
62
63     public static void main(String[] args) {
64         Punto puntoA = new Punto(0, 0);
65         Punto puntoB = new Punto(5, 5);
66
67         Rectangulo rectA = new Rectangulo(puntoA, puntoB);
68         System.out.println("Perímetro del rectángulo A: " + rectA.CalcularPerimetro());//20
69
70         puntoA.setX(4);
71         puntoA.setY(4);
72
73         Rectangulo rectB = new Rectangulo(puntoA, puntoB);
74         System.out.println("Creo un nuevo rectángulo, pero cambia el Perímetro del anterior");
75         System.out.println("Perímetro del rectángulo A: " + rectA.CalcularPerimetro());//20
76         System.out.println("Perímetro del rectángulo B: " + rectB.CalcularPerimetro());//4
77
78     }
79 }

```

De esta manera, se devuelve un punto totalmente nuevo que podrá ser manipulado sin ningún temor por parte del código cliente de la clase pues es una copia para él.

27.7.3. Ejemplo 2.2.1

Intenta rescribir los constructores de la clase `Rectangulo` teniendo en cuenta ahora su nueva estructura de atributos (dos objetos de la clase `Punto`, en lugar de cuatro elementos de tipo `double`): 1. Un constructor sin parámetros (para sustituir al constructor por defecto) que haga que los valores iniciales de las esquinas del rectángulo sean (0,0) y (1,1). 2. Un constructor con cuatro

parámetros, `x1`, `y1`, `x2`, `y2`, que cree un rectángulo con los vértices (`x1`, `y1`) y (`x2`, `y2`). 3. Un constructor con dos parámetros, `punto1`, `punto2`, que rellene los valores iniciales de los atributos del rectángulo con los valores proporcionados a través de los parámetros. 4. Un constructor con dos parámetros, `base` y `altura`, que cree un rectángulo donde el vértice inferior derecho esté ubicado en la posición (0,0) y que tenga una base y una altura tal y como indican los dos parámetros proporcionados. 5. Un constructor copia.

POSIBLE SOLUCIÓN

Durante el proceso de creación de un objeto (constructor) de la clase contenedora (en este caso `Rectangulo`) hay que tener en cuenta también la creación (llamada a constructores) de aquellos objetos que son contenidos (en este caso objetos de la clase `Punto`).

En el caso del primer constructor, habrá que crear dos puntos con las coordenadas (0,0) y (1,1) y asignarlos a los atributos correspondientes (`vertice1` y `vertice2`):

```
1 public Rectangulo (){
2     this.vertice1= new Punto (0,0);
3     this.vertice2= new Punto (1,1);
4 }
```

Para el segundo constructor habrá que crear dos puntos con las coordenadas `x1`, `y1`, `x2`, `y2` que han sido pasadas como parámetros:

```
1 public Rectangulo (double x1, double y1, double x2, double y2){
2     this.vertice1= new Punto (x1, y1);
3     this.vertice2= new Punto (x2, y2);
4 }
```

En el caso del tercer constructor puedes utilizar directamente los dos puntos que se pasan como parámetros para construir los vértices del rectángulo.

Ahora bien, esto podría ocasionar un efecto colateral no deseado si esos objetos de tipo `Punto` son modificados en el futuro desde el código cliente del constructor (no sabes si esos puntos fueron creados especialmente para ser usados por el rectángulo o si pertenecen a otro objeto que podría modificarlos más tarde).

Por tanto, para este caso quizás fuera recomendable crear dos nuevos puntos a imagen y semejanza de los puntos que se han pasado como parámetros. Para ello tendrías dos opciones:

1. Llamar al constructor de la clase `Punto` con los valores de los atributos (`x`, `y`).
2. Llamar al constructor copia de la clase `Punto`, si es que se dispone de él.

Aquí tienes las dos posibles versiones:

Constructor que "extrae" los atributos de los parámetros y crea nuevos objetos:

```
1 public Rectangulo (Punto vertice1, Punto vertice2) {
2     this.vertice1= vertice1;
3     this.vertice2= vertice2;
4 }
```

Constructor que crea los nuevos objetos mediante el constructor copia de los parámetros:

```
1 public Rectangulo(Punto vertice1, Punto vertice2) {
2     this.vertice1 = new Punto(vertice1.getX(), vertice1.getY());
3     this.vertice2 = new Punto(vertice2.getX(), vertice2.getY());
4 }
```

En este segundo caso puedes observar la utilidad de los constructores de copia a la hora de tener que clonar objetos (algo muy habitual en las inicializaciones).

```
1 public Rectangulo (Punto vertice1, Punto vertice2) {
2     this.vertice1= new Punto (vertice1 );
3     this.vertice2= new Punto (vertice2 );
4 }
```

Para el caso del constructor que recibe como parámetros la base y la altura, habrá que crear sendos vértices con valores (0,0) y (0 + base, 0 + altura), o lo que es lo mismo: (0,0) y (base, altura).

```

1  public Rectangulo(double base, double altura) {
2      this.vertice1 = new Punto(0, 0);
3      this.vertice2 = new Punto(base, altura);
4  }

```

Quedaría finalmente por implementar el constructor copia:

```

1  public Rectangulo (Rectangulo r) {
2      this.vertice1= new Punto (r.obtenerVertice1());
3      this.vertice2= new Punto (r.obtenerVertice2());
4  }

```

En este caso nuevamente volvemos a clonar los atributos `vertice1` y `vertice2` del objeto `r` que se ha pasado como parámetro para evitar tener que compartir esos atributos en los dos rectángulos. Así ahora el método main que comprueba la clase `Rectangulo` funciona correctamente:

```

1  public static void main(String[] args) {
2      Punto puntoA = new Punto(0, 0);
3      Punto puntoB = new Punto(5, 5);
4
5      Rectangulo rectA = new Rectangulo(puntoA, puntoB);
6      System.out.println("Perímetro del rectángulo A: " + rectA.CalcularPerimetro());//20
7
8      puntoA.setX(4);
9      puntoA.setY(4);
10
11     Rectangulo rectB = new Rectangulo(puntoA, puntoB);
12     System.out.println("Creo un nuevo rectángulo, pero cambia el Perímetro del anterior");
13     System.out.println("Perímetro del rectángulo A: " + rectA.CalcularPerimetro());//20
14     System.out.println("Perímetro del rectángulo B: " + rectB.CalcularPerimetro());//4
15 }

```

27.7.4. Ejemplo 3.1

Imagina que también necesitas una clase `Profesor`, que contará con atributos como nombre, apellidos, fecha de nacimiento, salario y especialidad. ¿Cómo crearías esa nueva clase y qué atributos le añadirías?

Está claro que un `Profesor` es otra especialización de `Persona`, al igual que lo era `Alumno`, así que podrías crear otra clase derivada de `Persona` y así aprovechar los atributos genéricos (nombre, apellidos, fecha de nacimiento) que posee todo objeto de tipo `Persona`. Tan solo faltaría añadirle sus atributos específicos (salario y especialidad):

```

1  public class Profesor extends Persona {
2      String especialidad;
3      double salario;
4      ...
5  }

```

27.7.5. Ejemplo 3.2

Reescribe las clases `Alumno` y `Profesor` utilizando el modificador `protected` para sus atributos del mismo modo que se ha hecho para su superclase `Persona`. Clase `Alumno`. Se trata simplemente de añadir el modificador de acceso `protected` a los nuevos atributos que añade la clase.

```

1  public class Alumno extends Persona {
2      protected String grupo;
3      protected double notaMedia;
4      ...
5  }

```

1. Clase `Profesor`. Exactamente igual que en la clase `Alumno`.

```

1  public class Profesor extends Persona {
2      protected String especialidad;
3      protected double salario;
4      ...
5  }

```

27.7.6. Ejemplo 3.3

Dadas las clases `Alumno` y `Profesor` que has utilizado anteriormente, implementa métodos `get` y `set` en las clases `Alumno` y `Profesor` para trabajar con sus cinco atributos (tres heredados más dos específicos).

POSIBLE SOLUCIÓN

- Clase `Alumno`. Se trata de heredar de la clase `Persona` y por tanto utilizar con normalidad sus atributos heredados como si pertenecieran a la propia clase (de hecho se puede considerar que le pertenecen, dado que los ha heredado).

```

1 package UD08._06_Ejemplo_3_3;
2
3 import java.time.LocalDate;
4
5 public class Alumno extends Persona {
6     protected String grupo;
7     protected double notaMedia;
8
9     // Método getNombre
10    public String getNombre() {
11        return nombre;
12    }
13
14    // Método getApellidos
15    public String getApellidos() {
16        return apellidos;
17    }
18
19    // Método getFechaNacim
20    public LocalDate getFechaNacim() {
21        return this.fechaNacim;
22    }
23
24    // Método getGrupo
25    public String getGrupo() {
26        return grupo;
27    }
28
29    // Método getNotaMedia
30    public double getNotaMedia() {
31        return notaMedia;
32    }
33
34    // Método setNombre
35    public void setNombre(String nombre) {
36        this.nombre = nombre;
37    }
38
39    // Método setApellidos
40    public void setApellidos(String apellidos) {
41        this.apellidos = apellidos;
42    }
43
44    // Método setFechaNacim
45    public void setFechaNacim(LocalDate fechaNacim) {
46        this.fechaNacim = fechaNacim;
47    }
48
49    // Método setGrupo
50    public void setGrupo(String grupo) {
51        this.grupo = grupo;
52    }
53
54    // Método setNotaMedia
55    public void setNotaMedia(double notaMedia) {
56        this.notaMedia = notaMedia;
57    }
58 }
```

Si te fijas, puedes utilizar sin problema la referencia `this` a la propia clase con esos atributos heredados, pues pertenecen a la clase: `this.nombre`, `this.apellidos`, etc.

- Clase `Profesor`. Seguimos exactamente el mismo procedimiento que con la clase `Alumno`.

```

1 package UD08._06_Ejemplo_3_3;
2
3 import java.time.LocalDate;
4
5 public class Profesor extends Persona {
6     String especialidad;
7     double salario;
8
9     // Método getNombre
10    public String getNombre() {
11        return nombre;
12    }
13
14    // Método getApellidos
15    public String getApellidos() {
16        return apellidos;
17    }
18
19    // Método getFechaNacim
20    public LocalDate getFechaNacim() {
21        return this.fechaNacim;
22    }
23
24    // Método getEspecialidad
25    public String getEspecialidad() {
26        return especialidad;
27    }
28
29    // Método getSalario
30    public double getSalario() {
31        return salario;
32    }
33
34    // Método setNombre
35    public void setNombre(String nombre) {
36        this.nombre = nombre;
37    }
38
39    // Método setApellidos
40    public void setApellidos(String apellidos) {
41        this.apellidos = apellidos;
42    }
43
44    // Método setFechaNacim
45    public void setFechaNacim(LocalDate fechaNacim) {
46        this.fechaNacim = fechaNacim;
47    }
48
49    // Método setSalario
50    public void setSalario(double salario) {
51        this.salario = salario;
52    }
53
54    // Método setEspecialidad
55    public void setEspecialidad(String especialidad) {
56        this.especialidad = especialidad;
57    }
58 }

```

Una conclusión que puedes extraer de este código es que has tenido que escribir los métodos `get` y `set` para los tres atributos heredados, pero ¿no habría sido posible definir esos seis métodos en la clase base y así estas dos clases derivadas hubieran también heredado esos métodos? La respuesta es afirmativa y de hecho es como lo vas a hacer a partir de ahora. De esa manera te habrías evitado tener que escribir seis métodos en la clase `Alumno` y otros seis en la clase `Profesor`.

Recuerda

Así que recuerda: **se pueden heredar tanto los atributos como los métodos.**

Aquí tienes un ejemplo de cómo podrías haber definido la clase `Persona` para que luego se hubieran podido heredar de ella sus métodos (y no sólo sus atributos):

```

1 package UD08._06_Ejemplo_3_3;
2
3 import java.time.LocalDate;
4
5 public class Persona {
6     protected String nombre;
7     protected String apellidos;
8     protected LocalDate fechaNacim;
9
10    // Método getNombre
11    public String getNombre() {
12        return nombre;
13    }
14
15    // Método getApellidos
16    public String getApellidos() {
17        return apellidos;
18    }
19
20    // Método getFechaNacim
21    public LocalDate getFechaNacim() {
22        return this.fechaNacim;
23    }
24
25    // Método setNombre
26    public void setNombre(String nombre) {
27        this.nombre = nombre;
28    }
29
30    // Método setApellidos
31    public void setApellidos(String apellidos) {
32        this.apellidos = apellidos;
33    }
34
35    // Método setFechaNacim
36    public void setFechaNacim(LocalDate fechaNacim) {
37        this.fechaNacim = fechaNacim;
38    }
39 }
```

27.7.7. Ejemplo 3.3.1

Dadas las clases `Persona`, `Alumno` y `Profesor` que has utilizado anteriormente, implementa métodos `get` y `set` en la clase `Persona` para trabajar con sus tres atributos y en las clases `Alumno` y `Profesor` para manipular sus cinco atributos (tres heredados más dos específicos), teniendo en cuenta que los métodos que ya hayas definido para `Persona` van a ser heredados en `Alumno` y en `Profesor`.

1. Clase `Persona`.

```

1 package UD08._07_Ejemplo_3_3_1;
2
3 import java.time.LocalDate;
4
5 public class Persona {
6     protected String nombre;
7     protected String apellidos;
8     protected LocalDate fechaNacim;
9
10    // Método getNombre
11    public String getNombre() {
12        return nombre;
13    }
14
15    // Método getApellidos
16    public String getApellidos() {
17        return apellidos;
18    }
19
20    // Método getFechaNacim
21    public LocalDate getFechaNacim() {
22        return this.fechaNacim;
23    }
24
25    // Método setNombre
26    public void setNombre(String nombre) {
27        this.nombre = nombre;
28    }
29
30    // Método setApellidos
31    public void setApellidos(String apellidos) {
32        this.apellidos = apellidos;
33    }
34
35    // Método setFechaNacim
36    public void setFechaNacim(LocalDate fechaNacim) {
37        this.fechaNacim = fechaNacim;
38    }
39}

```

- Clase `Alumno`. Al heredar de la clase `Persona` tan solo es necesario escribir métodos para los nuevos atributos (métodos especializados de acceso a los atributos especializados), pues los métodos genéricos (de acceso a los atributos genéricos) ya forman parte de la clase al haberlos heredado.

```

1 package UD08._07_Ejemplo_3_3_1;
2
3 public class Alumno extends Persona {
4     protected String grupo;
5     protected double notaMedia;
6
7     // Método getGrupo
8     public String getGrupo() {
9         return grupo;
10    }
11
12    // Método getNotaMedia
13    public double getNotaMedia() {
14        return notaMedia;
15    }
16
17    // Método setGrupo
18    public void setGrupo(String grupo) {
19        this.grupo = grupo;
20    }
21
22    // Método setNotaMedia
23    public void setNotaMedia(double notaMedia) {
24        this.notaMedia = notaMedia;
25    }
26}

```

Aquí tienes una demostración práctica de cómo la herencia permite una reutilización eficiente del código, evitando tener que repetir atributos y métodos. Sólo has tenido que escribir cuatro métodos en lugar de diez.

- Clase `Profesor`. Seguimos exactamente el mismo procedimiento que con la clase `Alumno`.

```

1 package UD08._07_Ejemplo_3_3_1;
2
3 public class Profesor extends Persona {
4     String especialidad;
5     double salario;
6
7     // Método getEspecialidad
8     public String getEspecialidad() {
9         return especialidad;
10    }
11
12    // Método getSalario
13    public double getSalario() {
14        return salario;
15    }
16
17    // Método setSalario
18    public void setSalario(double salario) {
19        this.salario = salario;
20    }
21
22    // Método setEspecialidad
23    public void setEspecialidad(String especialidad) {
24        this.especialidad = especialidad;
25    }
26 }
```

27.7.8. Ejemplo 3.4

Dadas las clases `Persona`, `Alumno` y `Profesor` que has utilizado anteriormente, redefine el método `getNombre` para que devuelva la cadena "Alumno: ", junto con el nombre del alumno, si se trata de un objeto de la clase `Alumno` o bien "Profesor ", junto con el nombre del profesor, si se trata de un objeto de la clase `Profesor`.

1. Clase `Alumno`. Al heredar de la clase `Persona` tan solo es necesario escribir métodos para los nuevos atributos (métodos especializados de acceso a los atributos especializados), pues los métodos genéricos (de acceso a los atributos genéricos) ya forman parte de la clase al haberlos heredado. Esos son los métodos que se implementaron en el ejercicio anterior (`getGrupo`, `setGrupo`, etc.).

Ahora bien, hay que escribir otro método más, pues tienes que redefinir el método `getNombre` para que tenga un comportamiento un poco diferente al `getNombre` que se hereda de la clase base `Persona`:

```

1 // Método getNombre
2 @Override
3 public String getNombre (){
4     return "Alumno: " + this.nombre;
5 }
```

En este caso podría decirse que se "renuncia" al método heredado para redefinirlo con un comportamiento más especializado y acorde con la clase derivada.

1. Clase `Profesor`. Seguimos exactamente el mismo procedimiento que con la clase `Alumno` (redefinición del método `getNombre`).

```

1 // Método getNombre
2 @Override
3 public String getNombre() {
4     return "Profesor: " + this.nombre;
5 }
```

27.7.9. Ejemplo 3.5

Dadas las clases `Persona`, `Alumno` y `Profesor`, define un método `mostrar` para la clase `Persona`, que muestre el contenido de los atributos (datos personales) de un objeto de la clase `Persona`. A continuación, define sendos métodos `mostrar` especializados para las clases `Alumno` y `Profesor` que "amplíen" la funcionalidad del método `mostrar` original de la clase `Persona`. 1. Método `mostrar` de la clase `Persona`.

```

1 public void mostrar () {
2     SimpleDateFormat formatoFecha = new SimpleDateFormat("dd/MM/yyyy");
3     String Stringfecha= formatoFecha.format(this.fechaNacim.getTime());
4
5     System.out.printf ("Nombre: %s\n", this.nombre);
6     System.out.printf ("Apellidos: %s\n", this.apellidos);
7     System.out.printf ("Fecha de nacimiento: %s\n", Stringfecha);
8 }
```

1. Método mostrar de la clase `Profesor`. Llamamos al método mostrar de su clase padre (`Persona`) y luego añadimos la funcionalidad específica para la subclase `Profesor` : ````java public void mostrar () { super.mostrar (); // Llamada al método "mostrar" de la superclase

```
// A continuación mostramos la información "especializada" de esta subclase System.out.printf ("Especialidad: %s\n", this.especialidad); System.out.printf ("Salario: %7.2f euros\n", this.salario); } ````
```

2. Método mostrar de la clase `Alumno`. Llamamos al método mostrar de su clase padre (`Persona`) y luego añadimos la funcionalidad específica para la subclase `Alumno` :

```
1  public void mostrar () {
2      super.mostrar ();
3      // A continuación mostramos la información "especializada" de esta subclase
4      System.out.print ("Grupo: %s\n", this.grupo);
5      System.out.print ("Nota media: %5.2f\n", this.notaMedia);
6  }
```

27.7.10. Ejemplo 3.6

Escribe un constructor para la clase `Profesor` que realice una llamada al constructor de su clase base para inicializar sus atributos heredados. Los atributos específicos (no heredados) sí deberán ser inicializados en el propio constructor de la clase `Profesor`.

```
1  public Profesor (String nombre, String apellidos, GregorianCalendar fechaNacim, String especialidad, double salario){
2      super (nombre, apellidos, fechaNacim);
3      this.especialidad= especialidad;
4      this.salario= salario;
5  }
```

Puedes hacer lo mismo para la clase `Alumno` .

```
1  public Alumno(String nombre, String apellidos, LocalDate fechaNacim, String grupo, double notaMedia) {
2      super(nombre, apellidos, fechaNacim);
3      this.grupo = grupo;
4      this.notaMedia = notaMedia;
5  }
```

27.7.11. Ejemplo 4.1

Basándote en la jerarquía de clases de ejemplo (`Persona`, `Alumno`, `Profesor`), que ya has utilizado en otras ocasiones, modifica lo que consideres oportuno para que `Persona` sea, a partir de ahora, una clase abstracta (no instanciable) y las otras dos clases sigan siendo clases derivadas de ella, pero sí instanciables.

En este caso lo único que habría que hacer es añadir el modificador `abstract` a la clase `Persona` . El resto de la clase permanecería igual y las clases `Alumno` y `Profesor` no tendrían porqué sufrir ninguna modificación.

```
1  public abstract class Persona {
2      protected String nombre;
3      protected String apellidos;
4      protected LocalDate fechaNacim;
5      ...
6  }
```

A partir de ahora no podrán existir objetos de la clase `Persona` . El compilador generaría un error.

Localiza en la API de Java algún ejemplo de clase abstracta.

Existen una gran cantidad de clases abstractas en la API de Java. Aquí tienes un par de ejemplos:

- La clase `AbstractList` :

```
1  public abstract class AbstractList<E> extends AbstractCollection<E> implements List<E>
```

De la que heredan clases instanciable como `Vector` o `ArrayList` .

- La clase `AbstractSequentialList` :

```
1  public abstract class AbstractSequentialList<E> extends AbstractList<E>
```

Que hereda de `AbstractList` y de la que hereda la clase `LinkedList`

27.7.12. Ejemplo 4.2

Basándose en la jerarquía de clases `Persona`, `Alumno`, `Profesor`, crea un método abstracto llamado `mostrar` para la clase `Persona`. Dependiendo del tipo de persona (alumno o profesor) el método `mostrar` tendrá que mostrar unos u otros datos personales (habrá que hacer implementaciones específicas en cada clase derivada).

Una vez hecho esto, implementa completamente las tres clases (con todos sus atributos y métodos) y utilízalas en un pequeño programa de ejemplo que cree un objeto de tipo `Alumno` y otro de tipo `Profesor`, los rellene con información y muestre esa información en la pantalla a través del método `mostrar`.

Dado que el método `mostrar` no va a ser implementado en la clase `Persona`, será declarado como abstracto y no se incluirá su implementación:

```
1 protected abstract void mostrar();
```

Recuerda que el simple hecho de que la clase `Persona` contenga un método abstracto hace que la clase sea abstracta (y deberá indicarse como tal en su declaración):

```
1 public abstract class Persona {  
2 ...
```

En el caso de la clase `Alumno` habrá que hacer una implementación específica del método `mostrar` y lo mismo para el caso de la clase `Profesor`.

1. Método `mostrar` para la clase `Alumno`.

```
1 @Override  
2 public void mostrar() {  
3     DateTimeFormatter formato = DateTimeFormatter.ofPattern("d/MM/yyyy");  
4     String stringFecha = formato.format(this.fechaNacim);  
5     System.out.printf("Nombre: %s\n", this.nombre);  
6     System.out.printf("Apellidos: %s\n", this.apellidos);  
7     System.out.printf("Fecha de nacimiento: %s\n", stringFecha);  
8     // A continuación mostramos la información "especializada" de esta subclase  
9     System.out.printf("Grupo: %s\n", this.grupo);  
10    System.out.printf("Nota media: %.2f\n", this.notaMedia);  
11 }
```

1. Método `mostrar` para la clase `Profesor`.

```
1 @Override  
2 public void mostrar() {  
3     DateTimeFormatter formato = DateTimeFormatter.ofPattern("d/MM/yyyy");  
4     String stringFecha = formato.format(this.fechaNacim);  
5     System.out.printf("Nombre: %s\n", this.nombre);  
6     System.out.printf("Apellidos: %s\n", this.apellidos);  
7     System.out.printf("Fecha de nacimiento: %s\n", stringFecha);  
8     // A continuación mostramos la información "especializada" de esta subclase  
9     System.out.printf("Especialidad: %s\n", this.especialidad);  
10    System.out.printf("Salario: %.2f euros\n", this.salario);  
11 }
```

1. Programa de ejemplo de uso.

Un pequeño programa de ejemplo de uso del método `mostrar` en estas dos clases podría ser:

```
1 package UD08._12_Ejemplo_4_2;  
2  
3 import java.time.LocalDate;  
4  
5 public class EjemploUso {  
6     public static void main(String[] args) {  
7         // Declaración de objetos  
8         Alumno alumno;  
9         Profesor profesor;  
10        // Creación de objetos (llamada a constructores)  
11        alumno = new Alumno("Juan", "Torres", LocalDate.of(1990, 10, 6), "1DAW", 7.5);  
12        profesor = new Profesor("Antonio", "Campos", LocalDate.of(1970, 8, 15), "Informatica", 2000);  
13        // Utilización del método mostrar  
14        alumno.mostrar();  
15        profesor.mostrar();  
16    }  
17 }
```

La salida debe ser algo parecido a esto:

```

1  Nombre: Juan
2  Apellidos: Torres
3  Fecha de nacimiento: 6/10/1990
4  Grupo: 1DAW
5  Nota media: 7,50
6  Nombre: Antonio
7  Apellidos: Campos
8  Fecha de nacimiento: 15/08/1970
9  Especialidad: Informatica
10 Salario: 2000,00 euros

```

27.7.13. Ejemplo 5.2

Crea una interfaz en Java cuyo nombre sea `Imprimible` y que contenga algunos métodos útiles para mostrar el contenido de una clase:

1. Método `devolverContenidoString`, que crea un `String` con una representación de todo el contenido público (o que se decida que deba ser mostrado) del objeto y lo devuelve. El formato será una lista de pares "nombre=valor" de cada atributo separado por comas y la lista completa encerrada entre llaves: "`{<nombre_atributo_1>=<valor_atributo_1>, ..., <nombre_atributo_n>=<valor_atributo_n>}`".
2. Método `devolverContenidoArrayList`, que crea un `ArrayList` de `String` con una representación de todo el contenido público (o que se decida que deba ser mostrado) del objeto y lo devuelve.
3. Método `devolverContenidoHashMap`, similar al anterior, pero en lugar devolver en un `ArrayList` los valores de los atributos, se devuelve en una `HashMap` en forma de pares (`nombre`, `valor`).

Se trata simplemente de declarar la interfaz e incluir en su interior esos tres métodos:

```

1 package UD08._13_Ejemplo_5_2;
2
3 import java.util.ArrayList;
4 import java.util.HashMap;
5
6 public interface Imprimible {
7     String devolverContenidoString();
8     ArrayList devolverContenidoArrayList();
9     HashMap devolverContenidoHashMap();
10 }

```

El cómo se implementarán cada uno de esos métodos dependerá exclusivamente de cada clase que decida implementar esta interfaz.

27.7.14. Ejemplo 5.3

Haz que las clases `Alumno` y `Profesor` implementen la interfaz `Imprimible` que se ha escrito en el ejercicio anterior.

La primera opción que se te puede ocurrir es pensar que en ambas clases habrá que indicar que implementan la interfaz `Imprimible` y por tanto definir los métodos que ésta incluye: `devolverContenidoString`, `devolverContenidoHashMap` y `devolverContenidoArrayList`.

Si las clases `Alumno` y `Profesor` no heredaran de la misma clase habría que hacerlo obligatoriamente así, pues no comparten superclase y precisamente para eso sirven las interfaces: para implementar determinados comportamientos que no pertenecen a la estructura jerárquica de herencia en la que se encuentra una clase (de esta manera, clases que no tienen ninguna relación de herencia podrían compartir interfaz).

Pero en este caso podríamos aprovechar que ambas clases sí son subclases de una misma superclase (heredan de la misma) y hacer que la interfaz `Imprimible` sea implementada directamente por la superclase (`Persona`) y de este modo ahorrarnos bastante código. Así no haría falta indicar explícitamente que `Alumno` y `Profesor` implementan la interfaz `Imprimible`, pues lo estarán haciendo de forma implícita al heredar de una clase que ya ha implementado esa interfaz (la clase `Persona`, que es padre de ambas).

Una vez que los métodos de la interfaz estén implementados en la clase `Persona`, tan solo habrá que redefinir o ampliar los métodos de la interfaz para que se adapten a cada clase hija específica (`Alumno` o `Profesor`), ahorrándonos tener que escribir varias veces la parte de código que obtiene los atributos genéricos de la clase `Persona`.

1. Clase `Persona`.

Indicamos que se va a implementar la interfaz `Imprimible`:

```

1  public abstract class Persona implements Imprimible {
2  ...

```

Definimos el método `devolverContenidoHashMap` a la manera de como debe ser implementado para la clase Persona. Podría quedar, por ejemplo, así:

```

1  @Override
2  public HashMap devolverContenidoHashMap() {
3      // Creamos la HashMap que va a ser devuelta
4      HashMap contenido = new HashMap();
5      // Añadimos los atributos de la clase
6      DateTimeFormatter formato = DateTimeFormatter.ofPattern("d/MM/yyyy");
7      String stringFecha = formato.format(this.fechaNacim);
8      contenido.put("nombre", this.nombre);
9      contenido.put("apellidos", this.apellidos);
10     contenido.put("fechaNacim", stringFecha);
11     // Devolvemos la HashMap
12     return contenido;
13 }

```

Del mismo modo, definimos también el método `devolverContenidoArrayList`:

```

1  @Override
2  public ArrayList devolverContenidoArrayList() {
3      // Creamos la ArrayList que va a ser devuelta
4      ArrayList contenido = new ArrayList();
5      // Añadimos los atributos de la clase
6      DateTimeFormatter formato = DateTimeFormatter.ofPattern("d/MM/yyyy");
7      String stringFecha = formato.format(this.fechaNacim);
8      contenido.add(this.nombre);
9      contenido.add(this.apellidos);
10     contenido.add(stringFecha);
11     // Devolvemos la ArrayList
12     return contenido;
13 }

```

Y por último el método `devolverContenidoString`:

```

1  @Override
2  public String devolverContenidoString() {
3      DateTimeFormatter formato = DateTimeFormatter.ofPattern("d/MM/yyyy");
4      String stringFecha = formato.format(this.fechaNacim);
5      String contenido = "{" + this.nombre + ", " + this.apellidos + ", " + stringFecha + "}";
6      return contenido;
7 }

```

1. Clase Alumno .

Esta clase hereda de la clase Persona, de manera que heredará los tres métodos anteriores. Tan solo habrá que redefinirlos para que, aprovechando el código ya escrito en la superclase, se añada la funcionalidad específica que aporta esta subclase.

```

1  public class Alumno extends Persona {
2  ...

```

Como puedes observar no ha sido necesario incluir el `implements Imprimible`, pues el `extends Persona` lo lleva implícito dado que Persona ya implementaba ese interfaz. Lo que haremos entonces será llamar al método que estamos redefiniendo utilizando la referencia a la superclase `super`.

El método `devolverContenidoHashMap` podría quedar, por ejemplo, así:

```

1  @Override
2  public HashMap devolverContenidoHashMap() {
3      // Llamada al método de la superclase
4      HashMap contenido = super.devolverContenidoHashMap();
5      // Añadimos los atributos específicos de la clase
6      contenido.put("grupo", this.grupo);
7      contenido.put("notaMedia", this.notaMedia);
8      // Devolvemos la HashMap rellena
9      return contenido;
10 }

```

1. Clase Profesor . En este caso habría que proceder exactamente de la misma manera que con la clase Alumno: redefiniendo los métodos de la interfaz `Imprimible` para añadir la funcionalidad específica que aporta esta subclase, en este caso mostraremos la redifinición del método `devolverContenidoArrayList()`:

```

1  @Override
2  public ArrayList devolverContenidoArrayList() {
3      // Llamada al método de la superclase
4      ArrayList contenido = super.devolverContenidoArrayList();
5      // Añadimos los atributos específicos de la clase
6      contenido.add(this.especialidad);
7      contenido.add(this.salario);
8      // Devolvemos la ArrayList
9      return contenido;
10 }

```

y la redefinición del método `devolverContenidoString()`:

```

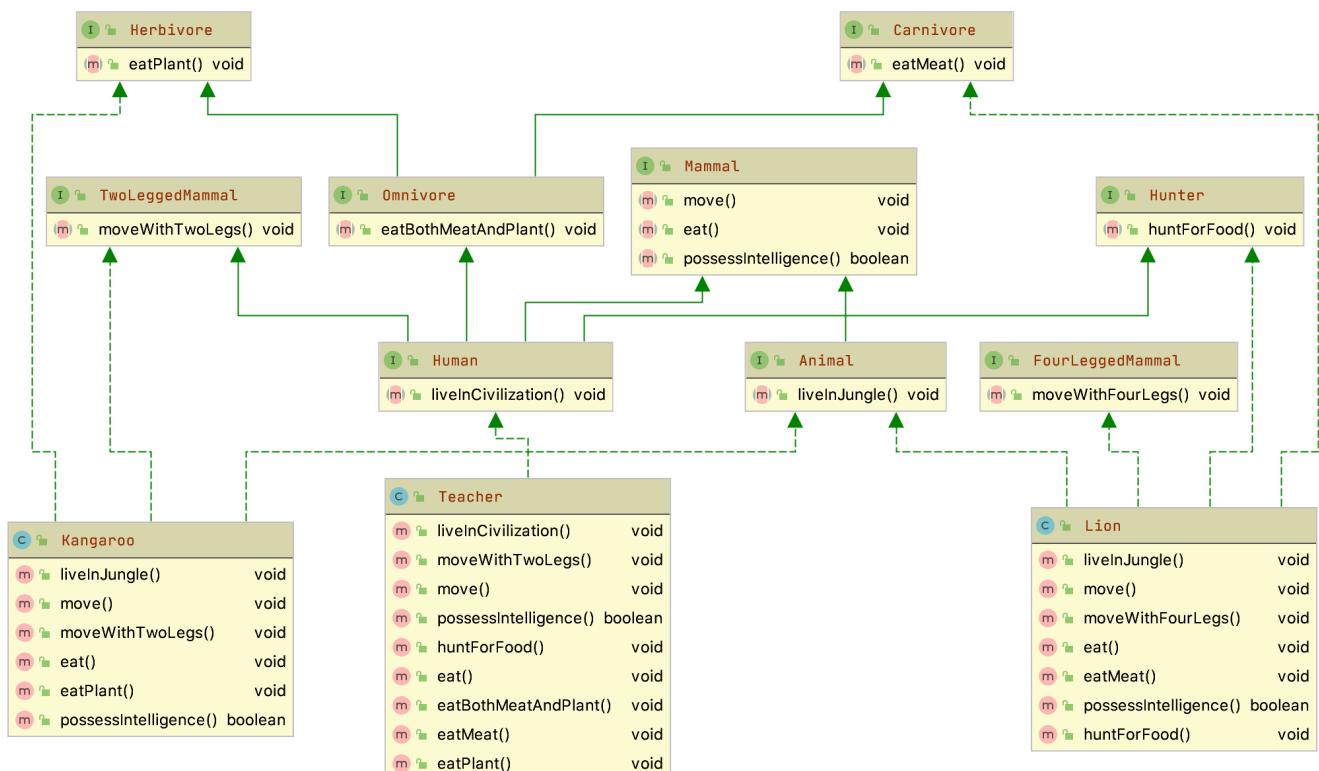
1  @Override
2  public String devolverContenidoString() {
3      // Llamada al método de la superclase
4      String contenido = super.devolverContenidoString();
5      // Eliminamos el último carácter, que contiene una llave de cierre.
6      contenido = contenido.substring(0, contenido.length() - 1);
7      contenido = contenido + ", " + this.especialidad + ", " + this.salario + "}";
8      // Devolvemos el String creado.
9      return contenido;
10 }

```

27.7.15. Ejemplo 5.4

¿Puede una clase implementar varias interfaces diferentes a la vez?

Observa el siguiente esquema UML:



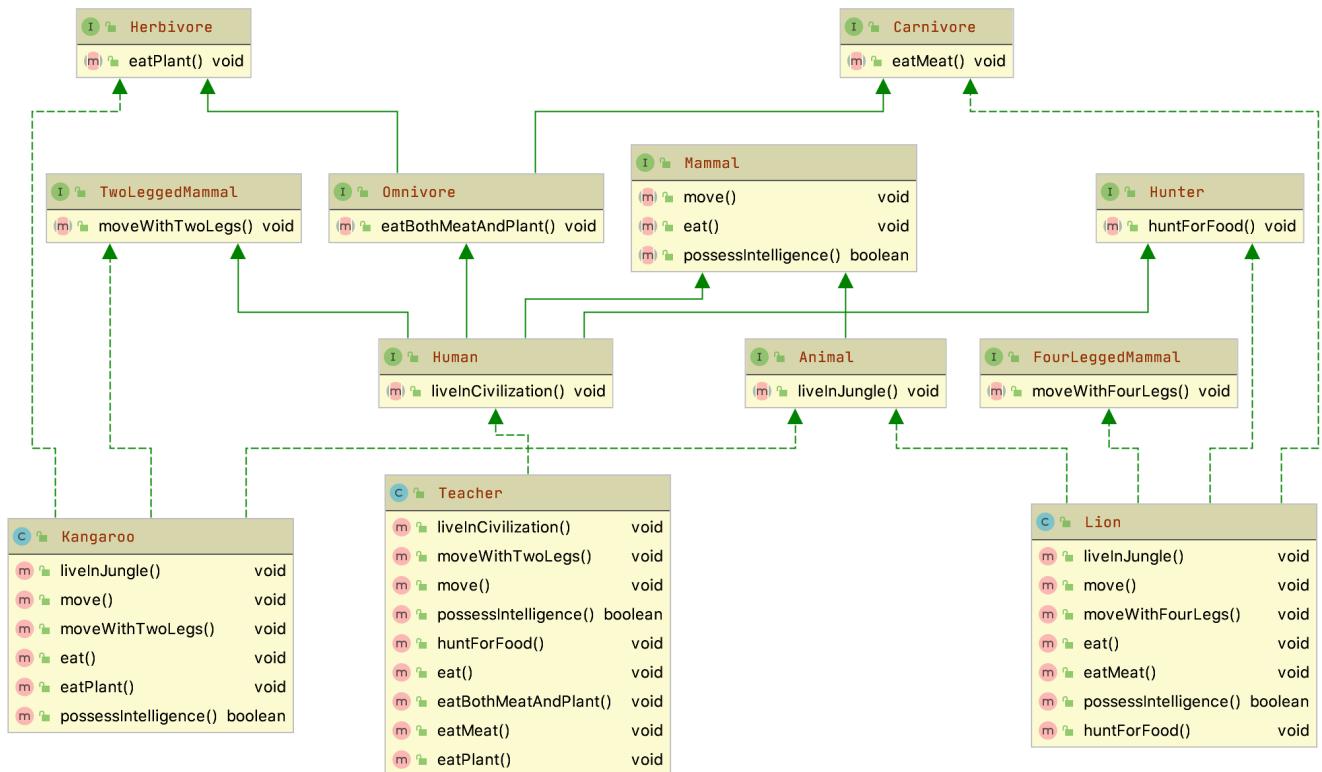
Las clases `Kangaroo` y `Lion` implementan varias clases:

- `Kangaroo`: `Herbivore`, `TwoLeggedMammal` y `Animal`
- `Lion`: `Animal`, `FourLeggedMammal`, `Hunter` y `Carnivore`

27.7.16. Ejemplo 5.5

¿Puede una interfaz heredar de varias interfaces diferentes a la vez?

Observa el siguiente esquema UML:



Las interfaces `Human` y `Omnivore` heredan de varias interfaces:

- `Human`: de `TwoLeggedMammal`, `Omnivore`, `Mammal` y `Hunter`
- `Omnivore`: `Herbivore` y `Carnivore`

27.7.17. Ejemplo 5.6

Supongamos una situación en la que nos interesa dejar constancia de que ciertas clases deben implementar una funcionalidad teórica determinada, diferente en cada clase afectada. Estamos hablando, pues, de la definición de un método teórico que algunas clases deberán implementar.

Un ejemplo real puede ser el método `calculoImporteJubilacion()` aplicable, de manera diferente, a muchas tipologías de trabajadores y, por tanto, podríamos pensar en diseñar una clase `Trabajador` en que uno de sus métodos fuera `calculoImporteJubilacion()`. Esta solución es válida si estamos diseñando una jerarquía de clases a partir de la clase `Trabajador` de la que cuelgan las clases correspondientes a las diferentes tipologías de trabajadores (metalúrgicos, hostelería, informáticos, profesores...). Además, disponemos del concepto de clase abstracta que cada subclase implemente obligatoriamente el método `calculoImporteJubilacion()`.

Pero, ¿y si resulta que ya tenemos las clases `Profesor`, `Informatico`, `Hostelero` en otras jerarquías de clases? La solución consiste en hacer que estas clases derivaran de la clase `Trabajador`, sin abandonar la derivación que pudieran tener, sería factible en lenguajes orientados a objetos que soportaran la herencia múltiple, pero esto no es factible en el lenguaje Java.

Para superar esta limitación, Java proporciona las interfaces.

Definición

Una interfaz es una **maqueta** contenedora de una lista de métodos abstractos y datos miembro (de tipos primitivos o de clases). Los atributos, si existen, son implícitamente consideradas `static` y `final`. Los métodos, si existen, son implícitamente considerados `public`.

Para entender en qué nos pueden ayudar las interfaces, necesitamos saber:

- Una interfaz puede ser implementada por múltiples clases, de manera similar a como una clase puede ser superclase de múltiples clases.
- Las clases que implementan una interfaz están obligadas a sobrescribir todos los métodos definidos en la interfaz. Si la definición de alguno de los métodos a sobrescribir coincide con la definición de algún método heredado, este desaparece de la clase.
- Una clase puede implementar múltiples interfaces, a diferencia de la derivación, que sólo se permite una única clase base.
- Una interfaz introduce un nuevo tipo de dato, por la que nunca habrá ninguna instancia, pero sí objetos usuarios de la interfaz (objetos de las clases que implementan la interfaz). Todas las clases que implementan una interfaz son compatibles con el tipo introducido por la interfaz.
- Una interfaz no proporciona ninguna funcionalidad a un objeto (ya que la clase que implementa la interfaz es la que debe definir la funcionalidad de todos los métodos), pero en cambio proporciona la posibilidad de formar parte de la funcionalidad de otros objetos (pasándola por parámetro en métodos de otras clases).
- La existencia de las interfaces posibilita la existencia de una jerarquía de tipo (que no debe confundirse con la jerarquía de clases) que permite la herencia múltiple.
- Una interfaz no se puede instanciar, pero sí se puede hacer referencia.

Importante

Así, si `I` es una interfaz y `C` es una clase que implementa la interfaz, se pueden declarar referencias al tipo `I` que apunten objetos de `C`:

```
1 I obj = new C (<parámetros>);
```

Las interfaces pueden heredar de otras interfaces y, a diferencia de la derivación de clases, pueden heredar de más de una interfaz.

Así, si diseñamos la interfaz `Trabajador`, podemos hacer que las clases ya existentes (`Profesor`, `Informatico`, `Hostelero` ...) la implementen y, por tanto, los objetos de estas clases, además de ser objetos de las superclases respectivas, pasan a ser considerados objetos usuarios del tipo `Trabajador`. Con esta actuación nos veremos obligados a implementar el método `calculoImporteJubilacion()` a todas las clases que implementen la interfaz.

Alguien no experimentado en la gestión de interfaces puede pensar: ¿por qué tanto revuelo con las interfaces si hubiéramos podido diseñar directamente un método llamado `calculoImporteJubilacion()` en las clases afectadas sin necesidad de definir ninguna interfaz?

La respuesta radica en el hecho de que la declaración de la interfaz lleva implícita la declaración del tipo `Trabajador` y, por tanto, podemos utilizar los objetos de todas las clases que implementen la interfaz en cualquier método de cualquier clase que tenga algún argumento referencia al tipo `Trabajador` como, por ejemplo, en un hipotético método de una hipotética clase llamada `Hacienda`:

```
1 public void enviarBorradorIRPF(Trabajador t) {...}
```

Por el hecho de existir la interfaz `Trabajador`, todos los objetos de las clases que la implementan (`Profesor`, `Informatico`, `Hostelero` ...) se pueden pasar como parámetro en las llamadas al método `enviarBorradorIRPF(Trabajador t)`.

La sintaxis para declarar una interfaz es:

```
1 [public] interface <NombreInterfaz> [extends <Nombreinterfaz1>, <Nombreinterfaz2>...] {
2     <CuerpoInterfaz>
3 }
```

Las interfaces también se pueden asignar a un paquete. La inexistencia del modificador de acceso público hace que la interfaz sea accesible a nivel del paquete.

Para los nombres de las interfaces, se aconseja seguir el mismo criterio que para los nombres de las clases.

Interfaces en cursiva

En la documentación de Java, las interfaces se identifican rápidamente entre las clases porque están en cursiva.

El cuerpo de la interfaz es la lista de métodos y/o constantes que contiene la interfaz. Para las constantes no hay que indicar que son `static` y `final` y para los métodos no hay que indicar que son `public`. Estas características se asignan implícitamente.

La sintaxis para declarar una clase que implemente una o más interfaces es:

```
1 [final] [public] class <NombreClase> [extends <NombreClaseBase>] implements <NombreInterfaz1>, <NomInterfaz2>... {
2     <CuerpoDeLaClase>
3 }
```

Los métodos de las interfaces a implementar en la clase deben ser obligatoriamente de acceso público.

Por último, comentar que, como por definición todos los datos miembro que se definen en una interfaz son `static` y `final`, y dado que las interfaz no se pueden instanciar, también resultan una buena herramienta para implantar grupos de constantes.

Así, por ejemplo:

```
1 public interface DiasSemana {
2     int LUNES = 1, MARTES=2, MIERCOLES=3, JUEVES=4;
3     int VIERNES=5, SABADO=6, DOMINGO=7;
4     String[] NOMBRES_DIAS = {"", "lunes", "martes", "miércoles", "jueves", "viernes", "sábado", "domingo"};
5 }
```

Esta definición nos permite utilizar las constantes declaradas en cualquier clase que implemente la interfaz, de manera tan simple como:

```
1 System.out.println (DiasSemana.NOMBRES_DIAS[LUNES]);
```

27.7.17.1. EJEMPLO DE DISEÑO DE INTERFAZ E IMPLEMENTACIÓN EN UNA CLASE

Se presentan un par de interfaces que incorporan datos (de tipo primitivo y de referencia en clase) y métodos y una clase que las implementa. En la declaración de la clase se ve que sólo implementa la interfaz `B`, pero como esta interfaz deriva de la interfaz `A` resulta que la clase está implementando las dos interfaces.

```
1 package UD08._17_Ejemplo_5_6;
2
3 import java.util.Date;
4
5 interface A {
6     Date ULTIMA_CREACION = new Date(0, 0, 1);
7     void metodoA();
8 }
9
10 interface B extends A {
11     int VALOR_B = 20;
12     // 1 -1 -1900
13     void metodoB();
14 }
15
16 public class Anexo5Interfaces implements B {
17     private long b;
18     private Date fechaCreacion = new Date();
19
20     public Anexo5Interfaces(int factor) {
21         b = VALOR_B * factor;
22         ULTIMA_CREACION.setTime(fechaCreacion.getTime());
23     }
24
25     @Override
26     public void metodoA() {
27         System.out.println("En metodoA, ULTIMA_CREACION = " + ULTIMA_CREACION);
28     }
29
30     @Override
31     public void metodoB() {
32         System.out.println("En metodoB, b = " + b);
33     }
34
35     public static void main(String args[]) {
36         System.out.println("Inicialmente, ULTIMA_CREACION = " + ULTIMA_CREACION);
37         Anexo5Interfaces obj = new Anexo5Interfaces(5);
38         obj.metodoA();
39         obj.metodoB();
40         A pa = obj;
41         B pb = obj;
42     }
43 }
```

Si lo ejecutamos obtendremos:

```
1 Inicialmente, ULTIMA_CREACION = Mon Jan 01 00:00:00 CET 1900
2 En metodoA, ULTIMA_CREACION = Thu Aug 26 16:09:47 CEST 2021
3 En metodoB, b = 100
```

El ejemplo sirve para ilustrar algunos puntos:

- Comprobamos que los datos miembro de las interfaces son `static`, ya que en el método `main()` hacemos referencia al dato miembro `ULTIMA_CREACION` sin indicar ningún objeto de la clase.
- Si hubiéramos intentado modificar los datos `VALOR_B` o `ULTIMA_CREACION` no habríamos podido porque es final, pero en cambio sí podemos modificar el contenido del objeto `Date` apuntado por `ULTIMA_CREACION`, que corresponde al momento temporal de la última creación de un objeto ya cada nueva creación se actualiza su contenido.
- En las dos últimas instrucciones del método `main()` vemos que podemos declarar variables `pa` y `pb` de las interfaces y utilizarlas para hacer referencia a objetos de la clase `EjemploInterfaz()`.

27.7.18. Ejemplo 6.2

Imagínate una clase que represente a instrumento musical genérico (`Instrumento`) y dos subclases que representen tipos de instrumentos específicos (por ejemplo `Flauta` y `Piano`). Todas las clases tendrán un método `tocarNota`, que será específico para cada subclase.

Haz un pequeño programa de ejemplo en Java que utilice el polimorfismo (referencias a la superclase que se convierten en instancias específicas de subclases) y la ligadura dinámica (llamadas a un método que aún no están resueltas en tiempo de compilación) con estas clases que representan instrumentos musicales. Puedes implementar el método `tocarNota` mediante la escritura de un mensaje en pantalla.

La clase `Instrumento` podría tener un único método (`tocarNota`):

```

1 package UD08._18_Ejemplo_6_2;
2
3 public abstract class Instrumento {
4     public void tocarNota(String nota) {
5         System.out.format("Instrumento: tocar nota %s.\n", nota);
6     }
7 }
```

En el caso de las clases `Piano` y `Flauta` puede ser similar, heredando de `Instrumento` y redefiniendo el método `tocarNota`:

```

1 package UD08._18_Ejemplo_6_2;
2
3 public class Flauta extends Instrumento {
4     @Override
5     public void tocarNota(String nota) {
6         System.out.format("Flauta: tocar nota %s.\n", nota);
7     }
8 }
```

```

1 package UD08._18_Ejemplo_6_2;
2
3 public class Piano extends Instrumento {
4     @Override
5     public void tocarNota(String nota) {
6         System.out.format("Piano: tocar nota %s.\n", nota);
7     }
8 }
```

Creamos una clase para comprobar su funcionamiento `EjemploUso`:

```

1 package UD08._18_Ejemplo_6_2;
2
3 import java.util.Scanner;
4
5 public class EjemploUso {
6     public static void main(String[] args) {
7         Scanner teclado = new Scanner(System.in);
8         System.out.print("Deseas un Piano o una Flauta (p o f)? ");
9         char respuesta = teclado.nextLine().toLowerCase().charAt(0);
10
11        Instrumento instrumento1; // Ejemplo de objeto polimórfico (podrá ser Piano o Flauta)
12        if (respuesta == 'p') {
13            // Ejemplo de objeto polimórfico (en este caso va adquirir forma de Piano)
14            instrumento1 = new Piano();
15        } else {
16            // Ejemplo de objeto polimórfico (en este caso va adquirir forma de Flauta)
17            instrumento1 = new Flauta();
18        }
19        // Interpretamos una nota con el objeto instrumento1
20        // No sabemos si se ejecutará el método tocarNota de Piano o de Flauta
21        // (dependerá de la ejecución)
22        instrumento1.tocarNota("do"); // Ejemplo de ligadura dinámica (tiempo de ejecución)
23    }
24 }
```

A la hora de declarar una referencia a un objeto de tipo instrumento, utilizamos la superclase (Instrumento):

```
1 Instrumento instrumento1; // Ejemplo de objeto polimórfico (podrá ser Piano o Flauta)
```

Sin embargo, a la hora de instanciar el objeto, utilizamos el constructor de alguna de sus subclases (Piano, Flauta, etc.):

```

1 if (respuesta == 'p') {
2     // Ejemplo de objeto polimórfico (en este caso va adquirir forma de Piano)
3     instrumento1 = new Piano();
4 } else {
5     // Ejemplo de objeto polimórfico (en este caso va adquirir forma de Flauta)
6     instrumento1 = new Flauta();
7 }
```

Finalmente, a la hora de invocar el método `tocarNota`, no sabremos a qué versión (de qué subclase) de `tocarNota` se estará llamando, pues dependerá del tipo de objeto (subclase) que se haya instanciado. Se estará utilizando por tanto la ligadura dinámica:

```

1 // Interpretamos una nota con el objeto instrumento1
2 // No sabemos si se ejecutará el método tocarNota de Piano o de Flauta
3 // (dependerá de la ejecución)
4 instrumento1.tocarNota("do"); // Ejemplo de ligadura dinámica (tiempo de ejecución)
```

27.7.19. Ejemplo 6.3

Haz un pequeño programa en Java en el que se declare una variable de tipo `Persona`, se pidan algunos datos sobre esa persona (nombre, apellidos y si es alumno o si es profesor), y se muestren nuevamente esos datos en pantalla, teniendo en cuenta que esa variable no puede ser instanciada como un objeto de tipo `Persona` (es una clase abstracta) y que tendrás que instanciarla como `Alumno` o como `Profesor`. Recuerda que para poder recuperar sus datos necesitarás hacer uso de la ligadura dinámica y que tan solo deberías acceder a métodos que sean de la superclase.

Si tuviéramos diferentes variables referencia a objetos de las clases `Alumno` y `Profesor` tendrías algo así:

```

1 Alumno obj1;
2 Profesor obj2;
3 ...
4 // Si se dan ciertas condiciones el objeto será de tipo Alumno y lo tendrás en obj1
5 System.out.printf ("Nombre: %s\n", obj1.getNombre());
6 // Si se dan otras condiciones el objeto será de tipo Profesor y lo tendrás en obj2
7 System.out.printf ("Nombre: %s\n", obj2.getNombre());
```

Pero si pudieras tratar de una manera más genérica la situación, podrías intentar algo así:

```

1 Persona obj;
2 // Si se dan ciertas condiciones el objeto será de tipo Alumno y por tanto lo instanciarás como tal
3 obj = new Alumno (<parámetros>);
4 // Si se otras condiciones el objeto será de tipo Profesor y por tanto lo instanciarás como tal
5 obj = new Profesor (<parámetros>);
```

De esta manera la variable `obj` podría contener una referencia a un objeto de la superclase `Persona` de subclase `Alumno` o bien de subclase `Profesor` (polimorfismo).

Esto significa que independientemente del tipo de subclase que sea (`Alumno` o `Profesor`), podrás invocar a métodos de la superclase `Persona` y durante la ejecución se resolverán como métodos de alguna de sus subclases:

```
1 //En tiempo de compilación no se sabrá de qué subclase de Persona será obj.
2 //Habrá que esperar la ejecución para que el entorno lo sepa e invoque al método adecuado.
3 System.out.format("Contenido del objeto: %s\n", obj.devolverContenidoString());
```

Por último recuerda que debes de proporcionar constructores a las subclases `Alumno` y `Profesor` que sean "compatibles" con algunos de los constructores de la superclase `Persona`, pues al llamar a un constructor de una subclase, su formato debe coincidir con el de algún constructor de la superclase (como debe suceder en general con cualquier método que sea invocado utilizando la ligadura dinámica).

Constructor "compatible" para `Alumno`:

```
1 public Alumno(String nombre, String apellidos, LocalDate fechaNacim) {
2     super(nombre, apellidos, fechaNacim);
3 }
```

y el constructor "compatible" para `Profesor`:

```
1 public Profesor(String nombre, String apellidos, LocalDate fechaNacim) {
2     super(nombre, apellidos, fechaNacim);
3 }
```

Aquí tienes el ejemplo completo de la clase `EjemploUso`:

```
1 package UD08._19_Ejemplo_6_3;
2
3 import java.time.LocalDate;
4 import java.util.Scanner;
5
6 public class EjemploUso {
7     public static void main(String[] args) {
8         Persona obj;
9         Scanner teclado = new Scanner(System.in);
10        System.out.print("Deseas crear un Profesor o un Alumno ('p' o 'a')?: ");
11        char respuesta = teclado.nextLine().toLowerCase().charAt(0);
12
13        if (respuesta == 'a') {
14            // Ejemplo de objeto polimórfico (en este caso va adquirir forma de Alumno)
15            obj = new Alumno("Alumno", "Apellidos", LocalDate.of(1977, 3, 8));
16        } else {
17            // Ejemplo de objeto polimórfico (en este caso va adquirir forma de Profesor)
18            obj = new Profesor("Profe", "Apellidos", LocalDate.of(1977, 3, 8));
19        }
20        System.out.format("Contenido del objeto: %s\n", obj.devolverContenidoString());
21    }
22 }
```

27.8. Píldoras informáticas relacionadas



⌚23 de febrero de 2026

28. 9.2 Ejercicios de la UD08

28.1. Ejercicios Herencia

1. Diseñar una jerarquía de clases para modelizar las **aulas de un centro de estudios**.

De un `Aula` se conoce el `código` (numérico), la `longitud` y la `anchura`. Se desea un método que devuelva la capacidad del aula sabiendo que esta se calcula a partir de la superficie a razón de 1 alumnos por cada 1.4 metros cuadrados de superficie.

Además de las aulas, digamos normales, existen aulas de informática y aulas de música. En las aulas de música se necesita conocer si tienen o no piano. De las aulas de informática se conoce el número de ordenadores y su capacidad no se calcula en función de la superficie, sino a razón de dos alumnos por ordenador.

Implementar el método `toString` de cada una de las clases diseñadas para que devuelva:

- En las aulas normales, el `código` y la `superficie` y la `capacidad`.
 - En las aulas de música e informática el texto irá precedido por "Aula de música" o "Aula de informática", según corresponda.
2. Un salón de **VideoJuegos** dispone de ordenadores en los que los clientes pueden jugar. Además de jugar en el establecimiento, la empresa alquila y vende juegos.

- a. Diseñar la clase `Juego` siguiendo las siguientes especificaciones:

- Atributos `protected`: `título` (`String`), `fabricante` (`String`), `año` (`int`).
- Constructor `public Juego(String t, String f, int a)`
- Consultores de todos los atributos
- `public String toString()`, que devuelve un `String` con los datos del Juego
- `public boolean equals(Object o)`: Dos juegos son iguales si tienen el mismo título, fabricante y año.
- `public int compareTo(Object o)`: Un juego es menor que otro si su título es menor. A igual título, si su fabricante es menor. A igual título y fabricante, si su año es menor.

- b. Diseñar las clases `JuegoEnAlquiler` y `JuegoEnVenta` (y otras si se considera oportuno), sabiendo que, además de los atributos descritos anteriormente, tienen.

- `precio`
- `nº de copias disponibles`
- `JuegoEnAlquiler`
- tiene un atributo que indica el número de días que se alquila. (Por el precio indicado, hay juegos que se alquilan por un dia, otros por 2, etc...)
- Constructor que recibe todos sus datos
- tiene un método `alquilar` que decrementa el número de copias disponibles.
- tiene un método `devolver` que incrementa el número de copias disponibles.
- `toString()` devuelve todos los datos del `JuegoEnAlquiler`
- `JuegoEnVenta`
- Constructor que recibe todos sus datos
- tiene un método `vender`, que decrementa el número de copias disponibles.
- `toString()` devuelve todos los datos del `JuegoEnVenta`

3. La **Fabrica Nacional de Moneda y Timbre** quiere almacenar cierta información técnica del dinero (billetes y monedas) que emite. En concreto, le interesa:

- `Valor`: Valor de la moneda o billete, en euros. (`double`)
- `Año de emisión`: Año en que fué emitida la moneda o billete. (`int`)
- De las monedas,
- `Diámetro`: Diámetro de la moneda, en milímetros. (`double`)
- `Peso`: Peso de la moneda, en gramos (`double`)
- De los billetes.
- `Altura del billete`, en mm (`double`)
- `Anchura del billete`, en mm (`double`).

- a. Diseñar la clase abstracta `Dinero` y sus subclases `Moneda` y `Billete`, desarrollando:

- Constructores que reciban los datos necesarios para inicializar los atributos de la clase correspondiente
 - `equals` : Dos monedas o billetes son iguales si tienen el mismo año de emisión y valor.
 - `compareTo` : Es menor (mayor) el de menor (mayor) año, a igual año es menor (mayor) el de menor (mayor) valor.
 - `toString` : Que muestre todos los datos del billete o moneda. Los billetes irán precedidos por el texto "BILLETE" y las monedas por el texto "MONEDA"
- b. Diseñar la clase `TestDinero` para probar las clases desarrolladas: Crear objetos de las clases `Moneda` y `Billete` y mostrarlos por pantalla.
4. Un **centro comercial** quiere mostrar cierta información sobre los televisores que vende. Los televisores pueden ser de dos tipos: de tubo o LCD. En concreto, de cada televisor le interesa mostrar
- Marca (`String`)
 - Modelo (`String`)
 - Precio en euros
 - Pulgadas de la pantalla (`double`).
 - Resolución: La resolución se mide de forma distinta en los televisores de tubo que en los televisores LCD.
 - En los TV de tubo se mide en lineas.
 - En los TV LCD se mide pixels horizontales x pixels verticales.
- a. Diseñar la clase `Televisor` con los atributos y métodos comunes a los dos tipos de televisores y sus subclases `TVTubo` y `TVLCD` con los atributos y métodos que sea necesario:
- Constructor de cada clase que permita inicializar todos los datos de la clase.
 - `equals` : Dos televisiones son iguales si son de la misma marca y modelo.
 - `compareTo` : Se considera menor (mayor) la de menor (mayor) marca. A igual marca, menor (mayor) la de menor (mayor) modelo.
 - `public String resolucion()` : Devuelve un texto con la resolución del televisor, como por ejemplo "420 lineas" o "800 x 600 pixels" dependiendo del tipo de televisor.
 - `public String toString()` : Devuelve un texto con la marca, modelo, precio, pulgadas y resolución.
- b. Diseñar la clase `TestTV` para probar las clases diseñadas. Crear algunos objetos de las clases `TVTubo` y `TVLCD` y mostrarlos por pantalla.
5. De cada pareja de afirmaciones **indica cual es la verdadera**:
- Pareja 1
 - Se dice que instanciamos una clase cuando creamos objetos de dicha clase.
 - Se dice que instanciamos una clase cuando creamos una subclase de dicha clase.
 - Pareja 2
 - Si una clase es abstracta no se puede instanciar.
 - Si una clase es abstracta no se puede heredar de ella.
 - Pareja 3
 - Una clase abstracta tiene que tener métodos abstractos.
 - Una clase puede ser abstracta y no tener métodos abstractos.
 - Pareja 4
 - Si una clase tiene métodos abstractos tiene que ser abstracta.
 - Una clase puede tener métodos abstractos y no ser abstracta.
 - Pareja 5
 - Si una clase es abstracta sus subclases no pueden ser abstractas.
 - Una clase abstracta puede tener subclases que también sean abstractas.
 - Pareja 6
 - Si un método es abstracto en una clase, tiene que ser no abstracto en la subclase, o bien, la subclase tiene que ser también abstracta.
 - Si un método es abstracto en una clase, no puede ser abstracto en las subclases.
 - Pareja 7
 - Si un método se define final se tiene que reescribir en las subclases.
 - Si un método se define final no se puede reescribir en las subclases.
 - Pareja 8

- Una clase puede tener un método final y no ser una clase final.
- Si una clase tiene un método final tiene que ser una clase final.
- i. Pareja 9
 - Si una clase se define final no se pueden definir subclases de ella.
 - Si una clase se define final no se puede instanciar.
- j. Pareja 10
 - Un método definido final y abstract resultaría inútil, puesto que nunca se podría implementar en las subclases.
 - Un método definido final y abstract podría resultar útil.
- k. Pareja 11
 - Una clase definida final y abstract resultaría inútil, puesto que no se podría instanciar ni heredar de ella.
 - Una clase definida final y abstract podría resultar útil.

6. Dada las siguientes **definiciones de clases**:

```

1  public class Persona {
2      private String nombre;
3      private int edad;
4
5      public Persona (){
6          this.nombre = "";
7          this.edad = 0;
8      }
9      public Persona(String n, int e){
10         this.nombre = n;
11         this.edad = e;
12     }
13     public String toString(){
14         return "Nombre: " + nombre + "Edad " + edad;
15     }
16     public final String getNombre (){
17         return nombre;
18     }
19     public final int getEdad(){
20         return edad;
21     }
22 }
```

```

1  class Estudiante extends Persona {
2      private double creditos;
3
4      public Estudiante(String n, int e, double c){
5          super(n,e);
6          this.creditos = c;
7      }
8      public String toString(){
9          return super.toString() + "\nCreditos: " + creditos;
10     }
11 }
```

```

1  class Empleado extends Persona {
2      private double salario;
3
4      public Empleado(String n, int e, double s){
5          super(n,e);
6          this.salario = s;
7      }
8      public String toString(){
9          return "Nombre: "+ this.nombre +
10             "\nSalario: "+ this.salario;
11     }
12 }
```

```

1  class Test{
2      public static void main(String[] args) {
3          Estudiante e = new Estudiante("pepe",18,100);
4          System.out.println(e.toString());
5      }
6  }
```

Responde a las siguientes cuestiones justificando las respuestas.

1. ¿Es necesario el uso de `this` en el constructor de la clase `Estudiante`?
2. ¿Es necesario el uso de `super` en el método `toString` de la clase `Estudiante`?
3. Si quitásemos el constructor de la clase `Estudiante`, ¿daría un error de compilación la clase `Estudiante`?
4. En el método `toString` de la clase `Empleado`, ¿por qué es incorrecto el acceso que se hace al atributo `nombre`? ¿Cómo se tendría que definir `nombre` en la clase `Persona` para evitar el error?
5. ¿Qué consecuencia tiene que algunos métodos de la clase `Persona` se hayan definido `final`?
6. Si el método `toString` no se hubiera definido en ninguna de las tres clases, ¿daría error el `sout` del método `main`?