

FUNDAMENTOS DE PROGRAMACIÓN 2

Tópicos de Programación Orientada a Objetos

Bienvenido amigo lector al mundo de la programación usando el paradigma Orientado a Objetos, aquí encontrarás desde los fundamentos más básicos de la programación orientada a objetos hasta los conceptos y mecanismos avanzados.

Se busca enseñar a programar de manera práctica con ejemplos que muestren las características importantes del paradigma orientado a objetos y también busca enseñar a solucionar problemas de cualquier área de conocimiento y de la vida diaria, desarrollando el pensamiento algorítmico.

Se introduce al estudiante al Lenguaje de Modelado Unificado (UML) que nos permite modelar software antes de implementarlo, siguiendo las recomendaciones de la Ingeniería de Software para crear productos de calidad.

Se presenta el desarrollo de un módulo de un videojuego de estrategia, que nos guiará en el aprendizaje de los diferentes conceptos del paradigma orientado a objetos.

En la medida de lo posible el texto está de acuerdo al estándar del lenguaje de programación orientado a objetos Java y es independiente del entorno de desarrollo, pero usaremos el entorno Open Source NetBeans para las prácticas.

Quisiera instar a los estudiantes a que desarrollem su autoaprendizaje, que aprendan a aprender, lograr la maestría mediante la práctica y al lograrlo no existirán los límites... y que con actitud y con voluntad se pueden alcanzar los más osados objetivos...



FUNDAMENTOS DE PROGRAMACIÓN 2 - Tópicos de Programación Orientada a Objetos



FUNDAMENTOS DE PROGRAMACIÓN 2

Tópicos de Programación Orientada a Objetos



EDITORIAL
UNSA

Marco Aedo López
Eveling Castro Gutiérrez

FUNDAMENTOS DE PROGRAMACIÓN 2

**Tópicos de Programación
Orientada a Objetos**

FUNDAMENTOS DE PROGRAMACIÓN 2

Tópicos de Programación Orientada a Objetos

Marco Aedo López

Eveling Castro Gutiérrez



**EDITORIAL
UNSA**

FUNDAMENTOS DE PROGRAMACIÓN 2

Tópicos de Programación Orientada a Objetos

Autores:

© Marco Aedo López, Eveling Castro Gutiérrez

Editado en:

© Editorial UNSA

Universidad Nacional de San Agustín de Arequipa

Calle Paucarpata, Puerta 5, Área de Ingenierías

Teléfono: 215558

E-mail: editorial@unsa.edu.pe

Arequipa - Perú

Impreso en:

Águila Real Publicidad Integral SRL

RUC.: 20600311248

Calle Nueva 327 Of. 221-A 2do. Piso Cercado

Primera edición, Noviembre del 2021

Tiraje: 600 ejemplares

Hecho el Depósito Legal en la Biblioteca Nacional del Perú

Nº 2021-12300

ISBN: 978-612-5035-20-2

Son propiedad del autor, ninguna parte de esta obra puede ser reproducida o transmitida, mediante ningún sistema o método, electrónico o mecánico, incluyendo el fotocopiado, la grabación o cualquier sistema de recuperación sin permiso escrito del Autor.

IMPRESO EN PERÚ

DEDICATORIA

A mi Madre, ejemplo de ser humano y de profesional,
a Ella por su luz, compañía y complicidad,
y a la Guardia Espiritual+=2...

A mi viejo... porque yo también te quiero,
te admiro... y te extraño...
"because a legend never dies,
this ain't the last goodbye"

Marco Aedo López

A mis estudiantes...
"la cima de una montaña es el pie de la siguiente"
Robin Sharma

Eveling Castro Gutiérrez

ÍNDICE GENERAL

Presentación	X
Lista de Figuras	XI
Lista de Tablas	XIII
Prefacio	XIV
Agradecimientos	XV
Objetivos	XVI
Conceptos Preliminares	XVI
CAPÍTULO I ARREGLOS ESTÁNDAR (ARRAYS)	1
1.1. Motivación	1
1.2. Fundamentos de los arreglos	1
1.3. Terminología	3
1.4. Sintaxis	3
1.5. Acceso a elementos	3
1.6. Otras formas de creación: Lista inicializadora	4
1.7. Utilizando elementos individuales	4
1.8. Características de los arreglos	6
1.9. Errores típicos	7
1.10. Usando una variable para el tamaño	7
1.11. Atributo <code>length</code> de los arreglos	8
1.12. Arreglos de objetos	9
1.13. Más ejemplos	11
1.14. Paso de arreglos estándar a métodos	12
1.15. Histogramas	16
1.16. Buscando en un arreglo	17
1.17. Búsqueda Lineal	18
1.18. Búsqueda Binaria	18
1.19. Ordenando un arreglo	23
1.20. Ordenamiento de Burbuja	23
1.21. Ordenamiento por Selección	24
1.22. Otros métodos de ordenamiento	26
1.23. Arreglos bidimensionales	26
1.24. Declaración y creación de arreglos bidimensionales	27
1.25. Otra forma de creación de arreglos bidimensionales	28
CAPÍTULO 2 ARRAYLIST Y HASHMAP	32
2.1. Motivación	32
2.2. Generalidades	32
2.3. Diferencias entre los <code>ArrayList</code> y los arreglos estándar	33
2.4. Métodos de la clase <code>ArrayList</code>	34
2.5. Otras consideraciones sobre los <code>ArrayList</code>	37

2.6.	Usando tipos de datos primitivos con ArrayList	37
2.7.	Sentencia for-each	38
2.8.	Métodos con ArrayList	39
2.9.	ArrayList Bidimensionales	39
2.10.	ArrayList vs arreglos estándar	41
2.11.	HashMap	41
2.12.	Principales métodos de HashMap	43
CAPÍTULO 3 FUNDAMENTOS DE LA ORIENTACIÓN A OBJETOS		48
3.1.	Antecedentes	48
3.2.	Conceptos	50
3.3.	Orientación a Objetos en Java	52
3.4.	Atributos y métodos set y get	54
3.5.	Inicializando objetos con constructores	56
3.6.	Plantilla para definición de una clase	57
3.7.	Métodos booleanos	57
3.8.	Método toString	57
3.9.	Diagrama de clases de UML	58
3.10.	Retornando un objeto de un método	60
3.11.	La referencia this	62
3.12.	Constructores sobrecargados	64
3.13.	Valores por default de los atributos	66
3.14.	Ambigüedad de nombres de variables	66
3.15.	Atributos constantes	66
3.16.	Encadenamiento de llamadas a métodos (Method-Call Chaining)	67
3.17.	Atributos y métodos de clase	68
3.18.	Atributos constantes de clase	69
3.19.	Clases de utilidad	69
3.20.	Los miembros de una Clase	70
CAPÍTULO 4 AGREGACIÓN, COMPOSICIÓN Y HERENCIA		72
4.1.	Introducción	72
4.2.	Agregación y composición	72
4.3.	Herencia	77
4.4.	Llamada al constructor de una superclase	81
4.5.	Sobrescritura de métodos (Overriding)	81
4.6.	Comparando la agregación, composición y herencia	82
CAPÍTULO 5 HERENCIA, POLIMORFISMO Y TÓPICOS AVANZADOS		84
5.1.	La clase Object	84
5.2.	El método equals	85
5.3.	El método toString	86
5.4.	Métodos toString de las clases Wrapper	87
5.5.	Polimorfismo	87
5.6.	Enlace Dinámico (Dynamic Binding)	89
5.7.	Operador instanceof	89
5.8.	Polimorfismo con arreglos	90
5.9.	Métodos y clases abstract	93
5.10.	Interfaces	94
5.11.	Herencia vs interfaces	95
5.12.	Modificador de acceso protected	96

CAPÍTULO 6 PROGRAMACIÓN ORIENTADA A EVENTOS E INTERFACE	
GRÁFICA DE USUARIO (GUI)	98
6.1. Introducción	98
6.2. Fundamentos de la programación orientada a eventos	99
6.3. Clases y paquetes GUI	99
6.4. La clase JFrame	101
6.5. Componentes de Java	101
6.6. Componente JLabel	102
6.7. Componente JTextField	102
6.8. Listener de componentes	103
6.9. Clase Interface	105
6.10. Clases interiores (Inner classes)	105
6.11. Clases interiores anónimas	105
6.12. Componente JButton	106
6.13. Cuadro de diálogo JOptionPane	108
6.14. Clase Color	109
6.15. Otros Componentes	112
CAPÍTULO 7 ARCHIVOS	114
7.1. Fundamentos de Entrada/Salida (E/S)	114
7.2. Clases para E/S de archivos	115
7.3. Operaciones básicas para E/S de archivos	116
7.4. Salida en archivos de texto	116
7.5. Entrada desde archivos de texto	118
7.6. Formato de texto vs Formato binario	120
7.7. Salida en archivos binarios	121
7.8. Entrada desde archivos binarios	121
7.9. E/S en un archivo de objetos	122
CAPÍTULO 8 BASES DE DATOS	126
8.1. Definiciones	126
8.2. Bases de Datos	126
8.3. Sistema Gestor de Bases de Datos (SGBD)	126
8.4. Anatomía de una Base de Datos	127
8.5. Modelado de una base de datos	127
8.6. Caso de estudio	128
8.7. Lenguaje SQL	129
8.8. Data Management Language (DML)	130
8.9. Java y BD	131
ANEXO – VIDEOJUEGO DE ESTRATEGIA	133
BIBLIOGRAFÍA	139

PRESENTACIÓN

Bienvenido amigo lector al mundo de la programación de computadoras usando el Paradigma Orientado a Objetos, aquí encontrarás desde los fundamentos más básicos de la programación orientada a objetos hasta los conceptos y mecanismos avanzados.

Se busca enseñar a programar de manera práctica con ejemplos que muestren las características importantes del Paradigma Orientado a Objetos y también busca enseñar a solucionar problemas de cualquier área de conocimiento y de la vida diaria, desarrollando el pensamiento algorítmico.

Se introduce al estudiante al Lenguaje de Modelado Unificado (UML) que nos permite modelar software antes de implementarlo, siguiendo las recomendaciones de la Ingeniería de Software para crear productos de calidad.

Se presenta el desarrollo de un módulo de videojuego de estrategia, que nos guiará en el aprendizaje de los conceptos del Paradigma Orientado a Objetos.

En la medida de lo posible el texto está de acuerdo al estándar del lenguaje de programación orientado a objetos Java y es independiente del entorno de desarrollo, pero usaremos el entorno de desarrollo Open Source NetBeans para las prácticas.

Quisiera instar a los estudiantes a que desarrollen su autoaprendizaje, que aprendan a aprender, lograr la maestría mediante la práctica y al lograrlo no existirán los límites...y que con actitud y voluntad se pueden alcanzar los más osados objetivos.

LISTA DE FIGURAS

Figura 1. Ejemplo de arreglo	2
Figura 2. Ejemplo de arreglo y acceso a sus elementos	2
Figura 3. Arreglo en la memoria RAM	4
Figura 4. Arreglos: realidad vs conceptualización	7
Figura 5. Arreglo de objetos en la memoria RAM	10
Figura 6. Arreglo como argumento y parámetro	13
Figura 7. Ejemplo de Histograma	16
Figura 8. Ejemplo de Búsqueda Lineal	18
Figura 9. Ejemplo de arreglo ordenado donde se podrá hacer la búsqueda binaria	19
Figura 10. Ejemplo de búsqueda binaria exitosa. Primera pasada	19
Figura 11. Ejemplo de búsqueda binaria exitosa. Segunda pasada	20
Figura 12. Ejemplo de búsqueda binaria exitosa. Tercera pasada	20
Figura 13. Ejemplo de búsqueda binaria fallida. Primera pasada	21
Figura 14. Ejemplo de búsqueda binaria fallida. Segunda pasada	21
Figura 15. Ejemplo de búsqueda binaria fallida. Tercera pasada	22
Figura 16. Ejemplo de búsqueda binaria fallida. Cuarta pasada	22
Figura 17. Ejemplo de ordenamiento de burbuja: primera pasada	24
Figura 18. Ejemplo de ordenamiento por selección: arreglo original y ordenado (para 2 pasadas)	25
Figura 19. Estructura de un arreglo bidimensional y cómo se accede a cada elemento	26
Figura 20. Tablas de multiplicar de varios números con multiplicadores del 1 al 9	27
Figura 21. Arreglo bidimensional llamado "lista" de 2 filas y 5 columnas	27
Figura 22. Arreglo bidimensional de 4x5	28
Figura 23. Arreglo bidimensional en la memoria RAM	29
Figura 24. Suma de arreglos bidimensionales	29
Figura 25. Producto de arreglos bidimensionales	30
Figura 26. Salida del programa <code>Tiempos de Vuelo</code>	30
Figura 27. Ubicación del <code>ArrayList</code> en la API de Java	33
Figura 28. <code>ArrayList</code> de <code>String</code>	34
Figura 29. Diagrama de memoria del <code>ArrayList</code> "amigos"	34
Figura 30. Ubicación del <code>HashMap</code> en la API de Java	42
Figura 31. Estructura de <code>HashMap</code> de n entradas	42
Figura 32. <code>HashMap</code> de <code>String, String</code>	43
Figura 33. Clase y Objetos: molde de galleta vs galletas comestibles	49
Figura 34. Ventana generada por ejecución del ejemplo	50
Figura 35. Esquema de un programa orientado a objetos	50
Figura 36. Objetos "Auto" y clase de donde provienen. Representación UML de Clase	51

Figura 37. Representación UML de Clase, incluye atributos y métodos	51
Figura 38. Objetos de cierta clase	52
Figura 39. Estado de la memoria al crear el objeto <code>miCurso</code>	53
Figura 40. Estado de la memoria al crear el objeto <code>per1</code>	55
Figura 41. Plantilla de cualquier clase en Java	57
Figura 42. Diagrama de Clases de UML del ejemplo de los cursos	59
Figura 43. Diagrama de Clases de UML completo del ejemplo de los cursos	59
Figura 44. Estado de la memoria al entrar al método <code>simplifica</code>	62
Figura 45. Estado de la memoria al retornar al método <code>main</code>	62
Figura 46. Referencia <code>this</code>	62
Figura 47. Clasificación de todos los miembros de una Clase	70
Figura 48. Relación de dependencia entre clases	72
Figura 49. Relación de Agregación de UML	73
Figura 50. Relación de Composición de UML	74
Figura 51. Diagrama de Clases UML del ejemplo planteado	74
Figura 52. Una posible clasificación de los seres vivos	78
Figura 53. Relación de Herencia de clases en UML	78
Figura 54. Diagrama de clases en UML considerando la herencia	78
Figura 55. Diagrama de clases detallado en UML considerando la herencia	80
Figura 56. Diagrama de clases en UML considerando la herencia	82
Figura 57. Diagrama de clases considerando una jerarquía de la clase <code>Object</code>	84
Figura 58. Diagrama de clases considerando una jerarquía de clases de herencia	89
Figura 59. Arreglo de Mascotas que permite objetos de subclases	90
Figura 60. Diagrama de clases para el problema	91
Figura 61. Representación UML de una clase abstracta (cursiva)	93
Figura 62. Representación UML de una interface (estereotipo <code><<interface>></code>)	94
Figura 63. Programación Orientada a Eventos	99
Figura 64. Resultado de la ejecución del programa	100
Figura 65. Resultado de la ejecución del programa	103
Figura 66. Resultado al ingresar Juancito y hacer ENTER	104
Figura 67. Resultado de la ejecución del programa	107
Figura 68. Ventana generada con JOptionPane	109
Figura 69. Interface de la aplicación a crear	110
Figura 70. Ubicación de paquete <code>java.io</code> para E/S	114
Figura 71. Principales clases utilizadas para E/S según el tipo de archivo	115
Figura 72. Esquema de la anatomía de una base de datos	127
Figura 73. Ejemplo de tabla: Alumno	127
Figura 74. Fases de Diseño de una BD	128
Figura 75. Modelo ER del caso de estudio (Diseño Conceptual)	129
Figura 76. Modelo Relacional del caso de estudio (Diseño Lógico derivado del Modelo ER)	129
Figura 77. Diagrama de Clases UML del videojuego de estrategia	134

LISTA DE TABLAS

Tabla 1.	Valores por default de cada tipo de dato primitivo de un arreglo estándar	6
Tabla 2.	Clases wrapper (envoltorio) de Java	37
Tabla 3.	ArrayList versus arreglos estándar	41
Tabla 4.	Valores por default de atributos dado su tipo de dato	66
Tabla 5.	Principales constantes de Color	109

PREFACIO

Arequipa, en la última década, se ha convertido en un polo de desarrollo nacional en todo lo que corresponde al área de la Computación, tanto en la formación académica como en el desarrollo de software y la investigación. Sin embargo, todavía está en proceso de alcanzar el siguiente nivel, nuestra presencia es regional y hasta cierto punto nacional, pero se puede acceder incluso a nivel mundial si es que contamos con el material humano bien educado. Aprender a programar siguiendo el Paradigma Orientado a Objetos es un primer paso para todo desarrollador de software profesional y constituye una actividad fundamental en la construcción de cualquier producto de software de calidad. Este texto busca aportar en la formación de dichos recursos humanos para fomentar una industria arequipeña del software más afianzada a nivel regional, nacional e internacional.

Enseñar a programar utilizando el Paradigma Orientado a Objetos no es una tarea fácil, pero tampoco es algo muy complejo, todo se basa en la lógica y en entender ciertos conceptos fundamentales que nos permitirán realizar una programación de mayor calidad. Un factor muy importante es el enfoque de enseñanza que se le da al curso, hacerlo de manera muy abstracta puede complicar los objetivos, es por eso que el presente texto busca alcanzar los objetivos utilizando diferentes herramientas y medios para que los estudiantes desarrollen 2 aspectos: el pensamiento algorítmico para que entiendan, creen y comparén algoritmos, y en segundo lugar utiliza el Paradigma Orientado a Objetos, plasmado en el lenguaje de programación Java, para concretar la solución a problemas aplicando un lenguaje de programación orientado a objetos y ayudándose en el modelado de software y otras técnicas de la Ingeniería de Software.

El presente texto académico no es para principiantes, sino corresponde a un segundo curso de programación de computadoras. Se asume que el estudiante domina los fundamentos de programación tales como: tipos de dato, variables, sentencias condicionales y repetitivas, además de utilización de métodos y uso de clases estándar de Java. Para lograr nuestros objetivos, inicialmente aprenderemos a utilizar las clases estándar de Java, para luego aprender a crear y utilizar nuestras propias clases, constituyendo esta metodología la mejor forma de dominar este paradigma.

Debemos resaltar que el presente texto académico se ha elaborado en colaboración y ha sido utilizado en la capacitación de estudiantes y profesionales desde el 2013 en la Escuela Profesional de Ingeniería de Sistemas, en la Segunda Especialidad de Ingeniería Informática y en el Centro de Investigación CITESOFT, entidades de la Universidad Nacional de San Agustín de Arequipa, obteniéndose destacados resultados.

AGRADECIMIENTOS

A la Universidad Nacional de San Agustín de Arequipa por el financiamiento brindado para la publicación del texto académico "Fundamentos de Programación 2 – Tópicos de Programación Orientada a Objetos", seleccionado en el Concurso 2020: "Publicación de Libros y Textos Académicos", con contrato N° PLT-07-2021-UNSA.

A la Universidad Nacional de San Agustín de Arequipa por brindarnos la oportunidad de formar a nuevas generaciones.

A los colegas docentes por tomar como referencia este humilde texto académico, esperando satisfacer las expectativas puestas en él.

A la comunidad educativa por la aceptación que tendrá el presente texto académico y con la fe... que no será el último que escribiremos.

A los docentes y estudiantes de la "Escuela Profesional de Ingeniería de Sistemas", "Segunda Especialidad de Ingeniería Informática" y del "Centro de Investigación, Transferencia de Tecnología y Desarrollo de Software – CiteSoft" de la UNSA, por su colaboración en la elaboración del texto.

A nuestras familias, "todos los actos pequeños suman", gracias por ser el motivo de enfrentarnos día a día a nuevos retos!.

Finalmente agradecer a todos nuestros compañeros y amigos que a lo largo de la vida nos enseñaron mucho... con el ejemplo y con antiejemplo también...



OBJETIVOS DEL TEXTO

- Enseñar el Paradigma Orientado a Objetos para desarrollo de programas de calidad utilizando el Lenguaje de Programación Java
- Desarrollar el pensamiento algorítmico
- Enseñar las características y principios avanzados del Lenguaje de Programación Java: arreglos estándar, ArrayList, HashMap, archivos, programación orientada a eventos e interfaces gráficas, y almacenamiento en bases de datos
- Aplicar los conocimientos adquiridos para solucionar problemas reales en su carrera profesional y su vida diaria

CONCEPTOS PRELIMINARES

I. ¿Qué es un lenguaje de programación (LP)?

Es un lenguaje artificial diseñado para expresar instrucciones que pueden ser llevadas a cabo por máquinas como las computadoras.

II. ¿Por qué existen los LPs?

Porque sirven como intermediario para la comunicación entre el humano y la computadora. Las computadoras tienen el lenguaje binario (combinación de 0s y 1s, ejemplo: 01101) y el humano tiene sus propios lenguajes para comunicarse.

III. ¿Qué tipo de lenguaje de programación es Java?

Los LPs se pueden dividir en 3 categorías:

- Lenguaje Máquina
- Lenguaje Ensamblador (bajo nivel)
- Lenguaje de Alto Nivel

Java sería un LP de alto nivel.

IV. ¿Cuál es la diferencia entre algoritmo, programa y software?

- Algoritmo: secuencia lógica y ordenada de instrucciones a seguir para solucionar un problema cualquiera (no necesariamente usando computadoras)
- Programa: implementación de un algoritmo en un LP
- Software: son los programas + documentación (modelos, diagramas, especificaciones, documentos de ayuda, planes de prueba, etc.), todo como un producto

V. ¿Qué necesitamos para programar?

- Pensamiento Algorítmico: habilidad para entender, ejecutar, evaluar, comparar y crear algoritmos

- Conocer la Sintaxis del Lenguaje de Programación (ejemplo: Java, Python, C++)

VI. ¿Cuáles son los componentes de un algoritmo?

- La entrada: son los datos sobre los que el algoritmo opera para hallar el resultado
- El proceso: son los pasos que hay que seguir, utilizando la entrada
- La salida: es el resultado que entrega el algoritmo

VII. ¿Cuáles son los pasos para la creación de un programa?

- 1º Entender el problema.
- 2º Plantear la lógica.
- 3º Codificar el programa.
- 4º Traducir el programa a lenguaje máquina.
- 5º Probar el programa.
- 6º Desplegar el programa.

VIII. ¿Qué otros conceptos preliminares se deben dominar para empezar este curso?

- Conocer y aplicar lo que son los tipos de dato, variables, sentencias condicionales y repetitivas, además de utilización de métodos y uso de clases estándar de Java

LECTURAS INTRODUCTORIAS

Un buen enfoque para ingresar al mundo de la programación de computadoras es usando videojuegos. El artículo internacional “Aproximación orientada a Entornos Lúdicos para la primera sesión de CS1 - Una experiencia con nativos digitales”, presentado en el 15th Latin American and Caribbean Conference for Engineering and Technology (LACCEI) - International Multi-Conference for Engineering, Education, and Technology, United States 2017, y el artículo internacional “Experiencia en la enseñanza de Fundamentos de Programación Orientada a Objetos a través de la implementación de un Videojuego de Estrategia” presentado en el 17th LACCEI International Multi-Conference for Engineering, Education, and Technology: “Industry, Innovation, And Infrastructure for Sustainable Cities and Communities”, 24-26 July 2019, Jamaica; son artículos indexados en SCOPUS y constituyen una gran referencia de cómo empezar a programar computadoras y también de los fundamentos del Paradigma Orientado a Objetos, y pueden ser accedidos libremente en:

http://www.laccei.org/LACCEI2017-BocaRaton/full_papers/FP249.pdf

http://www.laccei.org/LACCEI2019-MontegoBay/full_papers/FP63.pdf

CAPÍTULO 1

ARREGLOS ESTÁNDAR (ARRAYS)

OBJETIVOS:

- Comprender la estructura de datos arreglo estándar
- Saber declarar, crear e inicializar arreglos y referirse a sus elementos
- Utilizar los arreglos para almacenar, ordenar y examinar listas y tablas de valores
- Comprender los arreglos de objetos
- Comprender el uso de arreglos como argumento de métodos
- Aplicar los arreglos en técnicas de búsqueda y ordenamiento básicas
- Comprender los arreglos bidimensionales
- Utilizar los arreglos en la resolución de problemas

1.1. Motivación

Si se le planteara el siguiente cuestionamiento: "Se desea almacenar las notas de 100 estudiantes ingresadas por teclado".

Primera aproximación (según lo conocido hasta ahora, declaramos 100 variables e ingresamos los datos):

```
int nota1, nota2, nota3, . . . , nota100;  
nota1 = scan.nextInt();  
nota2 = scan.nextInt();  
nota3 = scan.nextInt();  
. . .  
nota100 = scan.nextInt();
```

¿Qué pasaría si la cantidad de notas fuera mayor?

El código sería mucho más largo y se tendría que definir más variables, acción tediosa y susceptible a cometer errores. A continuación, veremos una mejor solución usando arreglos estándar o arrays.

1.2. Fundamentos de los arreglos

Un arreglo es una estructura de datos que almacena un grupo de datos relacionados.

Todos los datos almacenados en el arreglo son del mismo tipo de dato y tienen el mismo nombre, pero se diferencian por su índice (posición).

EJEMPLO:

Si su programa necesita almacenar 5 notas finales del curso de Fundamentos de Programación (FP), es mejor utilizar un arreglo de dimensión 5 que 5 variables diferentes. El arreglo se llamará `notasFP` y tendrá 5 elementos.

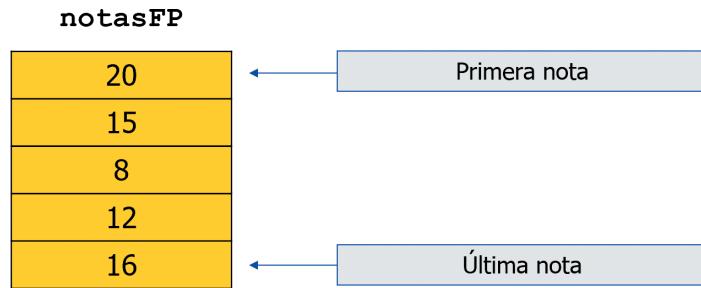


Figura 1. Ejemplo de arreglo

Un arreglo usa corchetes `[]` alrededor del índice para acceder a sus elementos específicos.

El número de índice empieza en **0** en lugar de 1, y el índice del último valor es igual a la **longitud – 1**.

EJEMPLO:

¿Cómo se accedería a cada elemento del arreglo estándar `notasFP`?

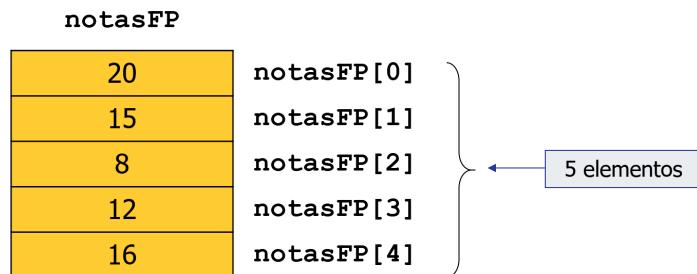


Figura 2. Ejemplo de arreglo y acceso a sus elementos

El arreglo se llama `notasFP`, tiene 5 elementos cuyos índices van de 0 a 4 y almacena 5 valores que se acceden usando el nombre del arreglo y el índice dentro de los corchetes. Cada elemento se puede utilizar como una variable común.

Se puede cambiar la segunda nota del arreglo:

```
notasFP[1] = 18;
```

Se quiere imprimir la primera nota del arreglo:

```
System.out.println(notasFP[0]);
```

Ahora se desea sumar las 2 primeras notas y almacenar la suma en una variable:

```
int suma = notasFP[0] + notasFP[1];
```

1.3. Terminología

- Un arreglo es una forma de lista ordenada.
- Está compuesto de **elementos**, donde cada uno tiene un índice y valor.
- Los elementos del arreglo tienen un **nombre común**.
- Los elementos del arreglo son referidos y diferenciados mediante su **índice**.
- El arreglo, como un todo, es referenciado a través de su nombre.
- El arreglo es **homogéneo**, esto es, los elementos son del mismo tipo de dato o **tipo base**.

1.4. Sintaxis

Un arreglo es un tipo de dato por referencia, por lo que antes de ser utilizado se debe declarar la variable de referencia del arreglo, luego crear el arreglo y finalmente asignarlo a la variable de referencia declarada.

Declaración:

Java acepta las siguientes formas de declarar arreglos, usaremos la primera.

```
<tipoDato> [ ] <nombreArreglo>;      //variación 1
<tipoDato>  <nombreArreglo> [ ];      //variación 2
```

Creación:

```
<nombreArreglo> = new <tipoDato> [<tamaño>];
```

Declaración y creación juntas:

```
<tipoDato>[ ] <nombreArreglo> = new <tipoDato> [<tamaño>];
```

EJEMPLO:

Indicar el código en Java para crear el arreglo estándar del ejemplo anterior.

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

O en una sola línea de código:

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

1.5. Acceso a elementos

Los elementos individuales en un arreglo son accedidos con la expresión de índice usando los corchetes [].

El índice en la primera posición del arreglo es **0** y en la última posición es **longitud-1**.

EJEMPLO:

```
double[ ] notas = new double[12];
notas[3] = 5;
```

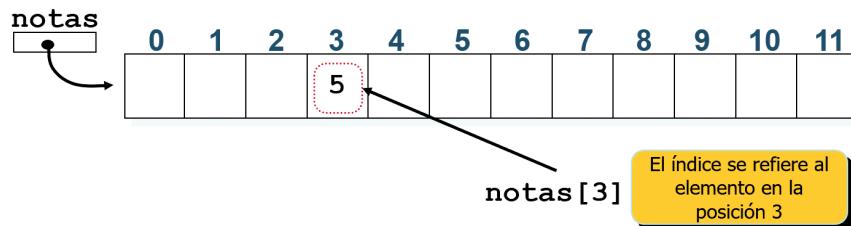


Figura 3. Arreglo en la memoria RAM

1.6. Otras formas de creación: Lista inicializadora

Un arreglo se puede crear e inicializar con ciertos valores a la vez.

EJEMPLO:

```
int[ ] numeros = {10, 20, 30, 40, 50, 60};
/* Arreglo unidimensional de 6 elementos enteros */
```

```
char[ ] c = {'T', 'o', 'n', 'i'};
/* Arreglo unidimensional de 4 elementos carácter */
```

1.7. Utilizando elementos individuales

Una vez creado el arreglo, cada elemento del mismo se accede de la siguiente forma:

```
nombreArreglo[índice]
```

Y se trata como una variable común en cualquier tipo de expresiones:

```
a[3] = a[1] + a[2];
a[4]++;
```

EJEMPLO:

```
double[ ] valores = new double[5];
for (int i = 0; i < 5; i++)
    valores[i] = 3.14*i;

double[ ] valores2 = new double[5];
for (int i = 0; i < valores2.length; i++)
    valores2[i] = scan.nextDouble();
```

EJEMPLO:

Siguiendo con el ejemplo de la motivación, quiero almacenar las notas de 100 estudiantes ingresadas por teclado.

```
Scanner scan = new Scanner(System.in);
int[] nota = new int[100];

for(int i=0;i<100;i++)
    nota[i] = scan.nextInt();
```

¿Qué pasaría si la cantidad de estudiantes fuera aún mayor?

El código sería fácilmente mantenible, en este caso sólo se cambiaría el literal 100.

EJEMPLO:

Encontrar el mínimo valor de 3 números enteros.

Veremos 3 versiones para solucionar el problema.

1. Sin usar arreglos

Considere el siguiente código:

```
int min, v1, v2, v3;

if ((v1 <= v2) && (v1 <= v3))
    min = v1;
else if ((v2 <= value1) && (v2 <= v3))
    min = v2;
else if ((v3 <= value1) && (v3 <= v2))
    min = v3;
```

¿Qué pasaría si la cantidad de números fuera mayor?

El código sería mucho más largo, la lógica para las condicionales if-elseif sería mucho más compleja.

2. Mejor solución

Considere el siguiente código:

```
int v1, v2, v3;
int min = v1;

if (v2 < min) min = v2;
if (v3 < min) min = v3;
```

¿Qué pasaría si la cantidad de números fuera mayor?

El código aún sería largo, pero podríamos incluir el patrón de comparaciones dentro de un bucle.

3. Usando arreglos

Considere el siguiente código:

```
int v[] = new int[3];
int min = v[0];
if (v[1] < min) min = v[1];
if (v[2] < min) min = v[2];
```

Las comparaciones podríamos insertarlas en un bucle:

```
int v[] = new int[3];
int min = v[0];

for (int i = 1; i < 3; i++) {
    if (v[i] < min)
        min = v[i];
}
```

¿Qué pasaría si la cantidad de números fuera mayor?

El código anterior puede ser fácilmente modificado para un arreglo mucho más largo.

Por ejemplo, si quisiéramos trabajar con 100 números enteros, sólo tendríamos que cambiar el literal 3 por 100 en la creación del arreglo y en el bucle.

1.8. Características de los arreglos

- El compilador reserva la cantidad de memoria apropiada.
- Los índices son denotados como expresiones en corchetes: []
- El tipo base del arreglo puede ser cualquier tipo: tipo de dato primitivo, tipo de dato por referencia; puede ser de un tipo propio de Java o definido por el usuario.
- El tamaño del arreglo puede ser especificado en tiempo de ejecución.
- El tipo del índice es entero y el rango de los índices va desde 0 a $n - 1$, donde n es el número de elementos del arreglo (longitud del arreglo).
- El control de no exceder los índices lo realiza el programador.
- Los arreglos **si tienen valores por default** para sus elementos, es 0 para los tipos numéricos y null para los tipos por referencia.
- length es un atributo que especifica el número de elementos del arreglo y se recomienda usarlo en lugar de usar literales.
- El arreglo es un objeto.
- Presenta las mismas características de todos los objetos.

Tipo Elemento	Valor por default
Entero	0
Real	0.0
Booleano	false
Por referencia (Objeto)	null

Tabla 1. Valores por default de cada tipo de dato primitivo de un arreglo estándar

EJEMPLO:

Comprobar que arreglos se crearon con el siguiente código.

```
char[] a;
int[] valores = new int[10];
String[] nombres = new String[2];
```

Se generaron:

- Arreglo a declarado, pero no creado, ni inicializado, no se puede usar aún
- Arreglo valores referencia a 10 nuevos elementos de tipo entero. Por defecto cada elemento se inicializa con 0
- Arreglo nombres referencia 2 elementos tipo String. Cada elemento es

inicializado por defecto con `null`

Por simplicidad, algunas veces conviene entender "el arreglo referenciado por la variable de referencia `valores`", como simplemente el "arreglo `valores`".

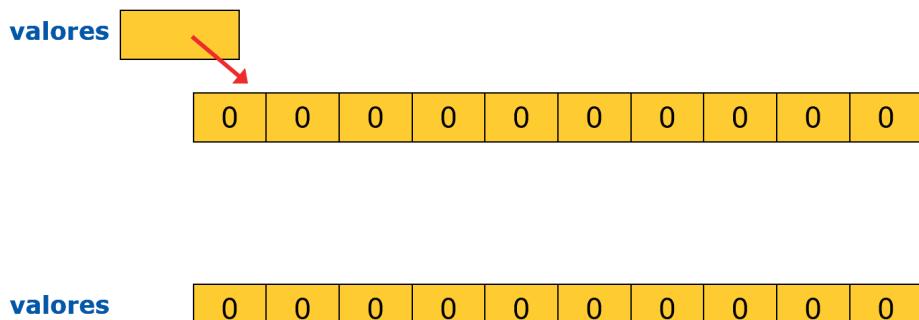


Figura 4. Arreglos: realidad vs conceptualización

1.9. Errores típicos

EJEMPLO:

Verificar los errores generados por el siguiente código.

```
int[] b = new int[500];
b[-1] = 123;
b[500] = 82;
```

- `b` de tipo arreglo hace referencia a una lista de 500 enteros.
- Cada elemento es inicializado en 0, entonces sus índices van de 0 a 499.
- Dos excepciones serán lanzadas:
 - -1 no es un índice válido – muy pequeño (error por uno)
 - 500 no es un índice válido – muy largo (error por uno)
- La excepción lanzada es `IndexOutOfBoundsException`.

1.10. Usando una variable para el tamaño

En Java, no estamos limitados a declarar un arreglo de tamaño fijo en el código. También se permite que el usuario ingrese el tamaño del arreglo en tiempo de ejecución.

EJEMPLO:

Hacer un programa que cree un arreglo de `n` enteros. El `n` se ingresa por teclado.

```
int n;
int[] numeros;

n = Integer.parseInt(JOptionPane.showInputDialog(null,
        "Tamaño del arreglo:"));
numeros = new int[n];
```

EJEMPLO:

Hacer un programa que cree un arreglo para n notas (el n se ingresa en tiempo de ejecución), luego que permita ingresarlas y finalmente que las imprima.

```
import java.util.*;

public class Ejemplo01fp {

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        int[] notasFP;
        int n;
        int nota;

        System.out.print("Cuantas notas desea ingresar?");
        n = scan.nextInt();
        notasFP = new int[n];
        for (int i=0; i<n; i++) {
            System.out.print("Ingresa nota: ");
            nota = scan.nextInt();
            notasFP[i] = nota;
        }

        System.out.println("Lista ingresada:");
        for (int i=0; i<n; i++)
            System.out.println((i + 1) + ". " + notasFP[i]);
    }
}
```

1.11. Atributo `length` de los arreglos

El atributo `length` de un arreglo permite obtener la cantidad de elementos que tiene el arreglo (su longitud). Su valor se actualiza automáticamente y por eso es preferible usar `length` que usar valores literales.

EJEMPLO:

Observar cómo el siguiente código utiliza el atributo `length` de cada arreglo.

```
int[] numeros = { 3, 5, 7, 8 };
double[] notas = { 12.41, 8.2, 1.3, 5.9, 8.2, 9, 15, 18.08 };
String[] meses = {"Enero", "Febrero", "Marzo", "Abril", "Mayo",
                  "Junio", "Julio", "Agosto", "Septiembre",
                  "Octubre", "Noviembre", "Diciembre" };

System.out.println(numeros.length);           //imprimirá 4
System.out.println(notas.length);            //imprimirá 8
System.out.println(meses.length);             //imprimirá 12
```

EJEMPLO:

Realizar un programa que cree un arreglo con capacidad para 100 números de celulares (que sólo acepte de 9 dígitos), que permita ingresar los necesarios hasta que ingrese una "q" y finalmente que imprima sólo los números ingresados.

```
import java.util.*;  
  
public class Ejemplo02fp {  
  
    public static void main(String[] args) {  
        Scanner scan = new Scanner(System.in);  
        String[] listaCell = new String[100];  
        int elementosIngresados = 0;  
        String numeroCell;  
  
        System.out.print("Entre numero de Celular (q para salir): ");  
        numeroCell = scan.next();  
        while (!numeroCell.equals("q") && elementosIngresados < listaCell.length) {  
            if (numeroCell.length() != 9)  
                System.out.println("Entrada invalida, debe ingresar un  
                numero de 9 digitos");  
            else{  
                listaCell[elementosIngresados] = numeroCell;  
                elementosIngresados++;  
            }  
            System.out.print("Entre numero de Celular (q para salir): ");  
            numeroCell = scan.next();  
        }  
  
        System.out.println("\nLista ingresada:");  
        for (int i=0; i<elementosIngresados; i++)  
            System.out.println((i + 1) + ". " + listaCell[i]);  
    }  
}
```

1.12. Arreglos de objetos

En Java, además de los arreglos de tipos de datos primitivos podemos declarar arreglos de objetos (tipos de datos por referencia).

Un arreglo de tipos de datos primitivos es una poderosa herramienta, pero un arreglo de objetos es mucho más poderoso aún.

El uso de un arreglo de objetos nos permite modelar las aplicaciones de una forma más limpia y lógica.

Los elementos de un arreglo de objetos serán referencias (direcciones de memoria) a otros objetos.

EJEMPLO:

Recordar que el tipo de dato `String` es un tipo de dato por referencia. Al crear un arreglo de `String` se estará creando un arreglo de objetos. Observar las 2 versiones equivalentes, en la primera se asigna los valores `String` directamente

como elementos del arreglo y en la segunda versión se crean los objetos String con new previamente.

```
public class Ejemplo03fp {

    public static void main(String[] args) {
        String[] misAmigos = new String[20];
        misAmigos[0] = "Juancito";
        misAmigos[1] = "Pedrito";

        for(int i=0;i<misAmigos.length;i++)
            System.out.println(misAmigos[i]);
    }
}
```

O su equivalente:

```
public class Ejemplo03fp {

    public static void main(String[] args) {
        String[] misAmigos = new String[20];
        misAmigos[0] = new String("Juancito");
        misAmigos[1] = new String("Pedrito");

        for(int i=0;i<misAmigos.length;i++)
            System.out.println(misAmigos[i]);
    }
}
```

En la siguiente figura se muestra el arreglo de objetos que se creó. misAmigos almacena la referencia del primer elemento del arreglo y cada elemento del arreglo almacena referencias a objetos de tipo String. En realidad, sólo los 2 primeros elementos referencian a objetos de tipo String ya creados, los otros elementos aún no referencian a ningún objeto y por eso tienen el valor null.

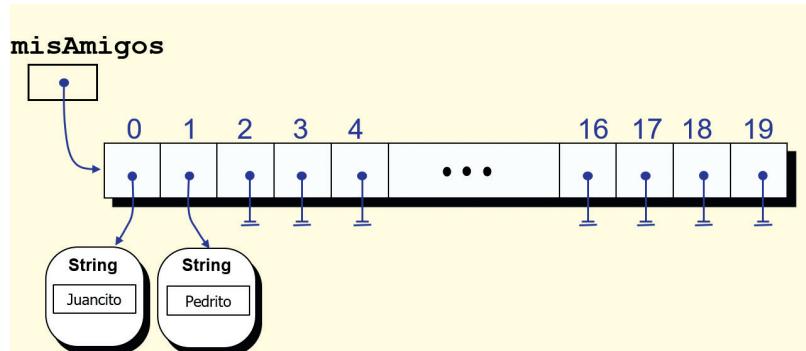


Figura 5. Arreglo de objetos en la memoria RAM

1.13. Más ejemplos

EJEMPLO:

Crear un programa que lea 5 notas enteras, almacenarlas en un arreglo y después mostrar el promedio sin redondear con el siguiente formato: “el promedio de n[0],n[1],n[2],n[3],n[4] es xx”.

```
import java.util.*;

public class Ejemplo04fp {

    public static void main(String[] args) {
        int[] n = new int[5];
        int total = 0;
        double promedio;
        Scanner scan = new Scanner(System.in);

        for(int i=0;i<n.length;i++){
            System.out.println("Ingrese un numero:");
            n[i] = scan.nextInt();
            total = total+n[i];
        }
        promedio = total*1.0/n.length;

        System.out.print("el promedio de ");
        for(int i=0;i<n.length;i++)
            System.out.print(n[i]+" ");
        System.out.println("es "+promedio);
    }
}
```

EJEMPLO:

Crear un arreglo de 10 enteros, asignarle como elementos los 10 primeros números impares positivos e imprimir sus elementos con sus índices, con formato:

```
Posición 0 valor 1
Posición 1 valor 3
...
Posición 9 valor 19
```

```
public class Ejemplo05fp {

    public static void main(String[] args) {
        int[] n = new int[10];

        for(int i=0;i<n.length;i++){
            n[i] = 2*i+1;
            System.out.println("Posicion "+i+" valor "+n[i]);
        }
    }
}
```

EJEMPLO:

Sumar 2 arreglos unidimensionales de tamaño 10 con valores aleatorios de notas que pertenezcan al rango [0..20] y almacenar la suma en un nuevo arreglo. Luego que imprima los valores de la suma con el formato: $x + y = z$

```
import java.util.*;

public class Ejemplo06fp {

    public static void main(String[] args) {
        final int tam=10;
        int[] a = new int[tam];
        int[] b = new int[tam];
        int[] c = new int[tam];
        Random rand = new Random();

        for(int i=0;i<a.length;i++){
            a[i] = rand.nextInt(21);
            b[i] = rand.nextInt(21);
            c[i] = a[i]+b[i];
            System.out.println(a[i]+“ + “+b[i]+“ = “+c[i]);
        }
    }
}
```

1.14. Paso de arreglos estándar a métodos

Los métodos también pueden aceptar argumentos de tipo arreglo. Un argumento siempre es pasado por valor, pero no se pasa toda una copia del arreglo, lo que se pasa es una copia del valor de la referencia del arreglo.

Con esa referencia y usando el parámetro correspondiente, se pueden alterar los valores del argumento original que se pasó al método.

EJEMPLO:

Observar cómo se crea un método que recibe como parámetro a un arreglo y cómo se invoca a dicho método con un argumento de tipo arreglo.

```
public class Ejemplo07fp {

    public static void main(String[] args) {

        double[] arregloUno = new double[20];
        minimo = busca(arregloUno);
    }

    public int busca(double[ ] misNumeros) {
        . . .
    }
}
```



La referencia (dirección) se copia:

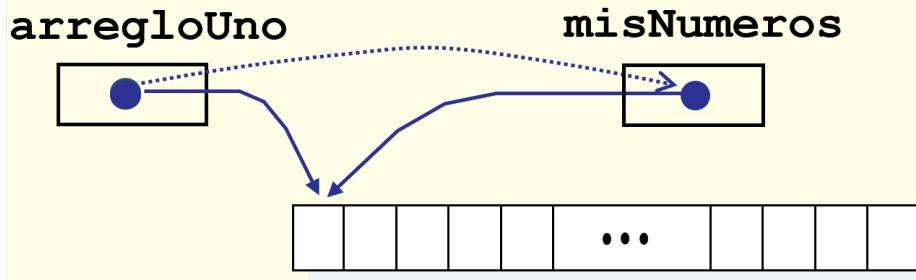


Figura 6. Arreglo como argumento y parámetro

Hay que considerar que la variable de referencia `arregloUno` es reconocida en el método `main` y la variable de referencia `misNumeros` es reconocida en el método `busca`, pero ambas referencian al mismo arreglo.

EJEMPLO:

Crear un arreglo utilizando una lista inicializadora y escribir un método `imprimir` que permita imprimir todos los elementos del arreglo.

```
public class Ejemplo08fp {

    public static void main(String[] args) {

        int[] a = {1,3,7,8};

        imprimir(a);
    }

    public static void imprimir(int[] b) {
        for(int i=0;i<b.length;i++)
            System.out.println(b[i]);
    }
}
```

EJEMPLO:

Crear un arreglo utilizando una lista inicializadora, un método `imprimir` que imprima todos los elementos del arreglo y un método `multiplicarX2` que altere todos los elementos del arreglo multiplicándolos por 2. Verificar que al cambiar el parámetro se cambió también el argumento (esto sólo sucede con los tipos de dato por referencia).

```

public class Ejemplo09fp {

    public static void main(String[] args) {

        int[] a = {1,3,7,8};

        imprimir(a);
        multiplicarx2(a);
        System.out.println("Ahora todos los elementos x2:");
        imprimir(a);
    }

    public static void imprimir(int[] b) {
        for(int i=0;i<b.length;i++)
            System.out.println(b[i]);
    }

    public static void multiplicarx2 (int[] b) {
        for(int i=0;i<b.length;i++)
            b[i]*=2;
    }
}

```

EJEMPLO:

Crear un programa para sumar 2 arreglos unidimensionales de tamaño 10 con valores de notas aleatorias enteras que pertenezcan al rango [0..20] y que permita almacenar la suma en un arreglo nuevo. Luego que imprima los valores de la suma con el formato:

x + y = z

Crear métodos `generar`, `sumar` e `imprimir`.

Observar los 2 programas creados, ambos son equivalentes. Especialmente prestar atención al método `sumar`, en el primer programa se aprovecha la propiedad que cualquier cambio en el parámetro del método afecta al argumento correspondiente (esto sólo se aplica para tipos de dato por referencia) y en el segundo programa el método `sumar` retorna un arreglo con la suma respectiva (observar específicamente la cabecera del método `sumar` y cómo se retorna un arreglo desde un método).

```

import java.util.*;

public class Ejemplo10fp {

    public static void main(String[] args) {
        final int tam = 10;
        int[] a = new int[tam];
        int[] b = new int[tam];
        int[] c = new int[tam];

        generar(a,b);
        sumar(a,b,c);
        imprimir(a,b,c);
    }
}

```

```

        }
    public static void generar(int[] x,int[] y) {
        Random rand = new Random();
        for(int i=0;i<x.length;i++) {
            x[i] = rand.nextInt(21);
            y[i] = rand.nextInt(21);
        }
    }
    public static void sumar(int[] x,int[] y,int[] z) {
        for(int i=0;i<x.length;i++)
            z[i] = x[i]+y[i];
    }
    public static void imprimir(int[] x, int[] y,int[] z) {
        for(int i=0;i<x.length;i++)
            System.out.println(x[i]+” + “+y[i]+” = “+z[i]);
    }
}

import java.util.*;

public class Ejemplo10fp {

    public static void main(String[] args) {
        final int tam = 10;
        int[] a = new int[tam];
        int[] b = new int[tam];
        int[] c;

        generar(a,b);
        c = sumar(a,b);
        imprimir(a,b,c);
    }

    public static void generar(int[] x,int[] y) {
        Random rand = new Random();
        for(int i=0;i<x.length;i++) {
            x[i] = rand.nextInt(21);
            y[i] = rand.nextInt(21);
        }
    }

    public static int[] sumar(int[] x,int[] y) {
        int[] z = new int[x.length];
        for(int i=0;i<x.length;i++)
            z[i] = x[i]+y[i];
        return z;
    }

    public static void imprimir(int[] x, int[] y,int[] z) {
        for(int i=0;i<x.length;i++)
            System.out.println(x[i]+” + “+y[i]+” = “+z[i]);
    }
}

```

EJERCICIO 1:

Crear un arreglo de n nombres de personas que permita almacenar datos que ingresemos para cada una. Luego que los muestre (mostrar solamente los datos de aquellas posiciones del arreglo que estén llenas).

TIP: llevar la cuenta del número de elementos llenados en el arreglo.

EJERCICIO 2:

Se tiene un arreglo con n precios que maneja la tienda en Enero, se quiere un nuevo arreglo para Febrero con los mismos valores de Enero, pero cambiando solamente el segundo valor a 13.99.

TIP: si asignamos una variable de referencia de un arreglo a otra variable de referencia de arreglo, ambas variables de referencia apuntarán al mismo arreglo. Si cambiamos algún valor del arreglo usando una de las variables, afectará el valor que referencia la otra variable.

1.15. Histogramas

Es un gráfico que muestra cantidades para un conjunto de categorías. Las muestra con barras, de mayor o menor tamaño que pueden ser fácilmente comparables.

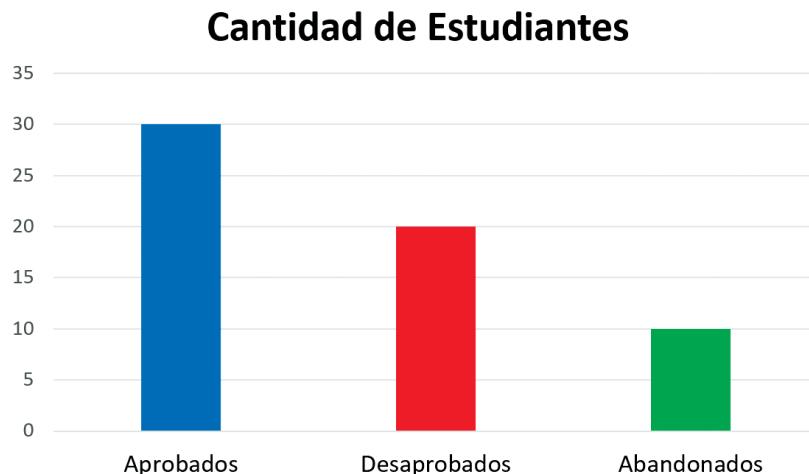


Figura 7. Ejemplo de Histograma

EJEMPLO:

Suponga que tiene 3 monedas, al lanzar las 3 monedas una cantidad de veces Ud. desea saber cuántas veces salieron 0 caras, 1 caras, 2 caras y 3 caras. En otras palabras, desea saber la probabilidad (distribución de frecuencia) para el número de caras.

Salida para simulación de 1 millón de lanzamientos:

Número de veces que cada caso ocurría (0, 1, 2, 3 caras):

```
0 124785 ****
1 374659 ****
2 375520 ****
3 125036 ****
```

```

public class Ejemplo11fp {

    public static void main(String[] args) {
        final int NUM_MONEDAS = 3;           // numero de monedas
        final int NUM_REPETICIONES = 1000000; // numero de
                                              lanzamientos
        int[] frecuencia;                  //almacena la frecuencia
        int caras;                        //numero de caras en lanzamiento
        double fraccionRepeticiones;
        int asteriscos;                  // numero de asteriscos

        frecuencia = new int[NUM_MONEDAS + 1]; //4 posibles casos
        for (int rep=0; rep<NUM_REPETICIONES; rep++) {
            caras = 0;
            for (int i=0; i<NUM_MONEDAS; i++)
                caras += (int) (Math.random() * 2); //1 es cara
            frecuencia[caras]++;
        }
        System.out.println("Número veces que contador de caras salio:");

        for (caras=0; caras<=NUM_MONEDAS; caras++) {
            System.out.print(" " + caras + " " + frecuencia[caras] + " ");
            fraccionRepeticiones = (double) frecuencia[caras]
                                      //NUM_REPETICIONES;
            asteriscos = (int) Math.round(fraccionRepeticiones * 100);
            for (int i=0; i<asteriscos; i++)
                System.out.print("*");
            System.out.println();
        }
    }
}

```

EJERCICIO 3:

Hacer un histograma y demostrar que el 7 es la suma que se obtiene con mayor probabilidad cuando realizamos el lanzamiento de 2 dados.

TIP: Recordar la teoría de probabilidades: para que dos dados sumen 2 hay una sola combinación (1,1), para sumar 12 también hay una sola combinación (6,6) y para que sumen 7 hay varias posibles combinaciones (1,6) (6,1) (2,5) (5,2) (3,4) (4,3). Simular el lanzamiento de dados una cantidad elevada de veces.

1.16. Buscando en un arreglo

Es frecuente la necesidad de determinar si un arreglo contiene un valor en particular.

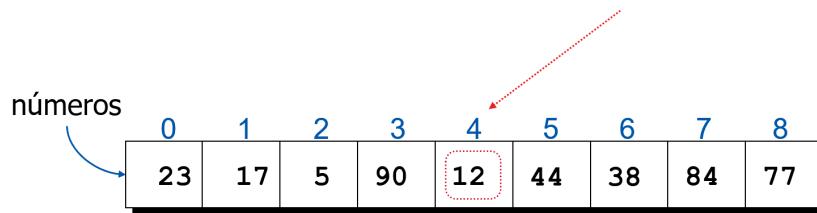
Cuando tenemos una colección de datos, una de las operaciones que necesitaremos es un método de **búsqueda** para localizar un elemento en dicha colección.

Definición del problema:

Dado un valor x devolver el índice de x en el arreglo, si tal x existe. Caso contrario, retornar -1 (NO_ENCONTRADO). Asumimos que retorna el índice de la primera vez que se encuentra el valor x.

1.17. Búsqueda Lineal

Realiza la búsqueda de un valor desde el inicio hasta el fin del arreglo, cuando encuentra el valor devuelve la posición o índice en que lo encontró, pero devuelve -1 si no lo encontró.



Búsqueda fallida: búsqueda (45) → -1 (NO_ENCONTRADO)

Búsqueda exitosa: búsqueda (12) → 4

Figura 8. Ejemplo de Búsqueda Lineal

EJEMPLO:

Realizar el seudocódigo del método de búsqueda lineal y su implementación en Java.

```

busquedaLineal(lista[], valor)
    para i desde 0 hasta lista.longitud-1 inc 1
        si lista[i]==valor
            retornar i
        finSi
    finCiclo
    retornar -1
FIN

public static int busquedaLineal(int[] lista, int valor) {
    for(int i=0;i<lista.length;i++)
        if(lista[i]==valor)
            return i;
    return -1;
}

```

1.18. Búsqueda Binaria

¿Y si el arreglo está ordenado?. No necesitaríamos recorrer todo el arreglo para encontrar un valor, porque llegará el momento que la búsqueda se hará inútil, ya que el arreglo está **ordenado** y tendremos la certeza de encontrarlo o no.

Aplicamos la Búsqueda Binaria. Asumiremos que el arreglo está ordenado ascendenteamente. Primero buscamos el valor en la posición media del arreglo. Si coincide con el elemento buscado x, se FINALIZA. Si el valor buscado x es menor al valor en la posición media, se busca en la mitad izquierda del arreglo. Si el valor buscado x es mayor al valor en la posición media, se busca en la mitad derecha del arreglo. Y ASÍ SE CONTINÚA hasta encontrarlo, si no lo encuentra devuelve -1 (NO_ENCONTRADO).

números

0	1	2	3	4	5	6	7	8
5	12	17	23	38	45	77	84	90

Figura 9. Ejemplo de arreglo ordenado donde se podrá hacer la búsqueda binaria

- 1) Búsqueda binaria exitosa

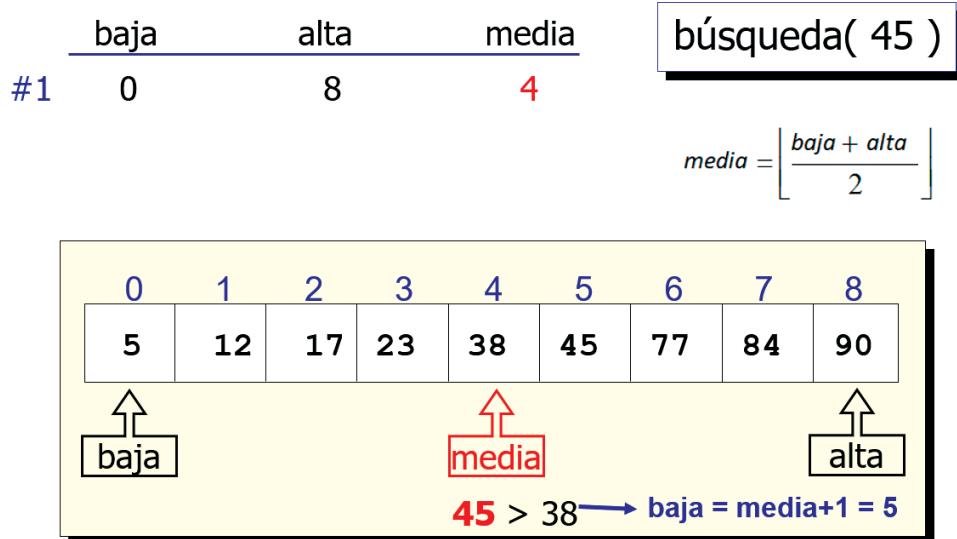


Figura 10. Ejemplo de búsqueda binaria exitosa. Primera pasada

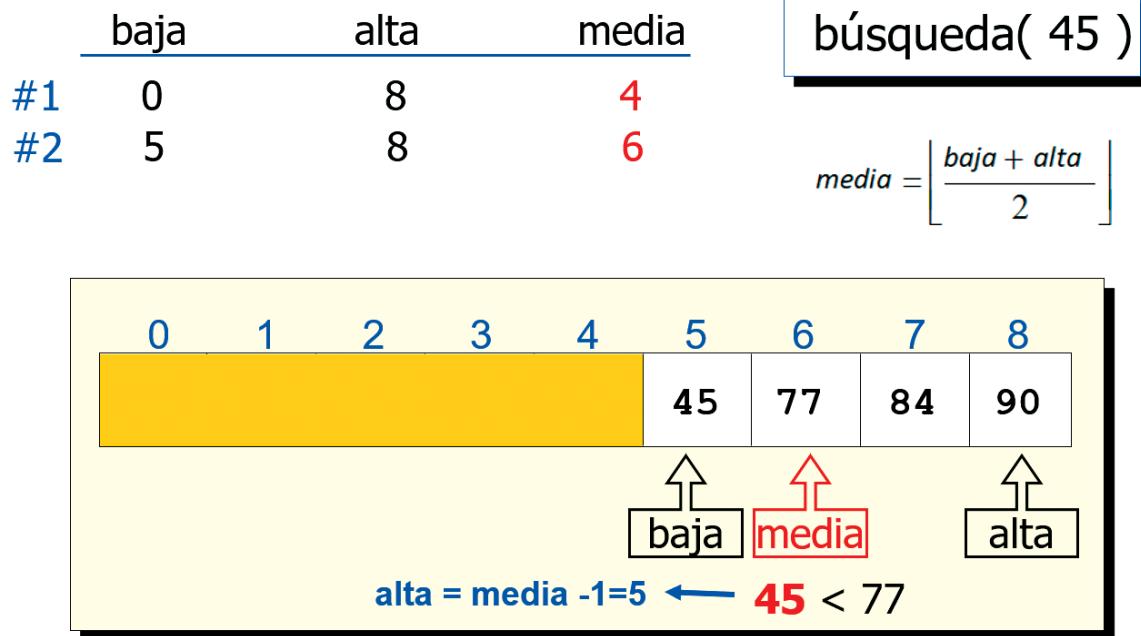


Figura 11. Ejemplo de búsqueda binaria exitosa. Segunda pasada

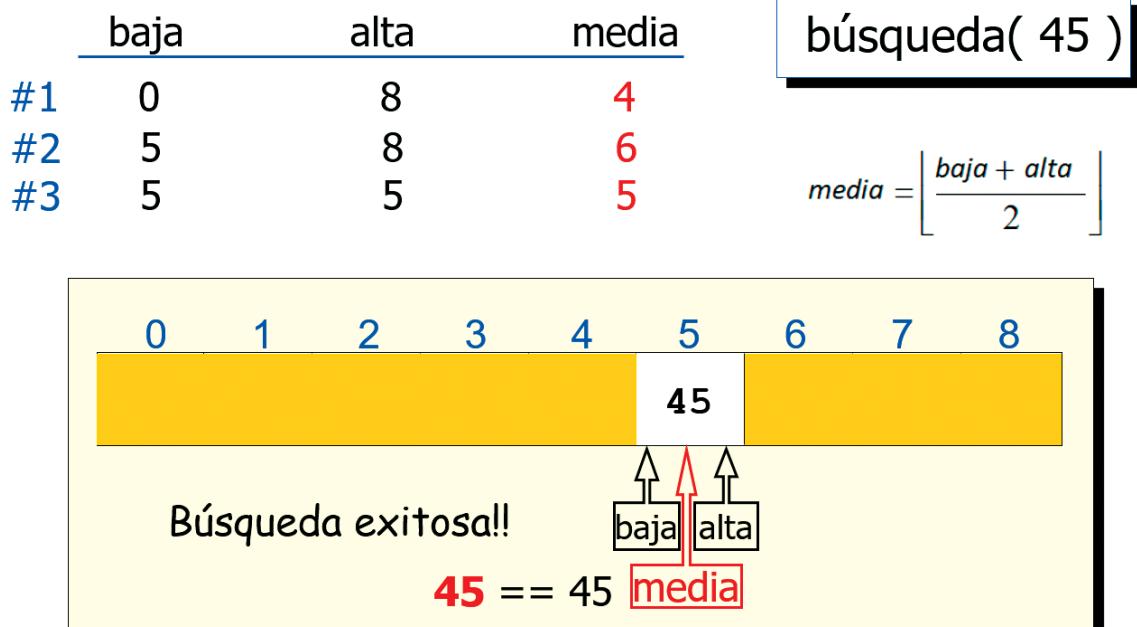


Figura 12. Ejemplo de búsqueda binaria exitosa. Tercera pasada

2) Búsqueda binaria fallida

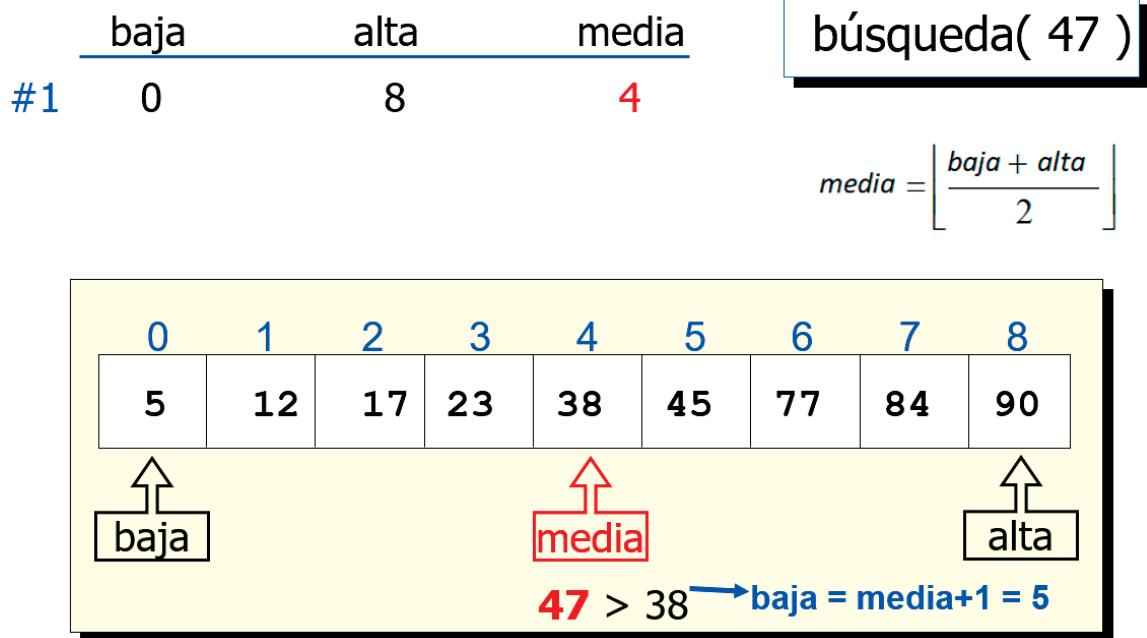


Figura 13. Ejemplo de búsqueda binaria fallida. Primera pasada

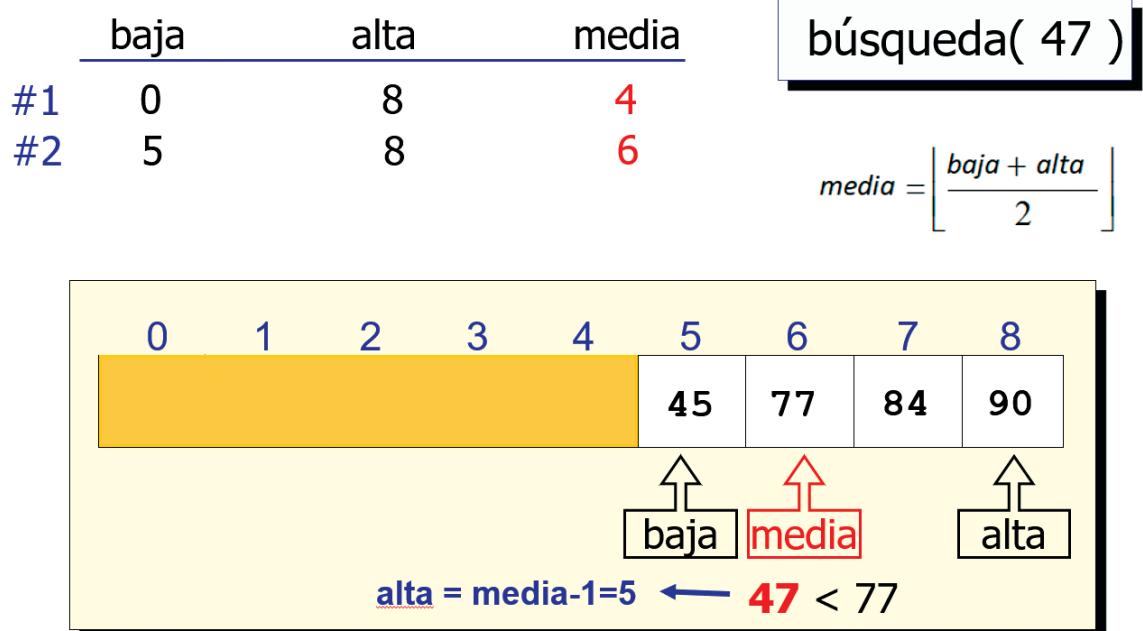


Figura 14. Ejemplo de búsqueda binaria fallida. Segunda pasada

	baja	alta	media	búsqueda(47)
#1	0	8	4	
#2	5	8	6	
#3	5	5	5	$media = \left\lfloor \frac{baja + alta}{2} \right\rfloor$

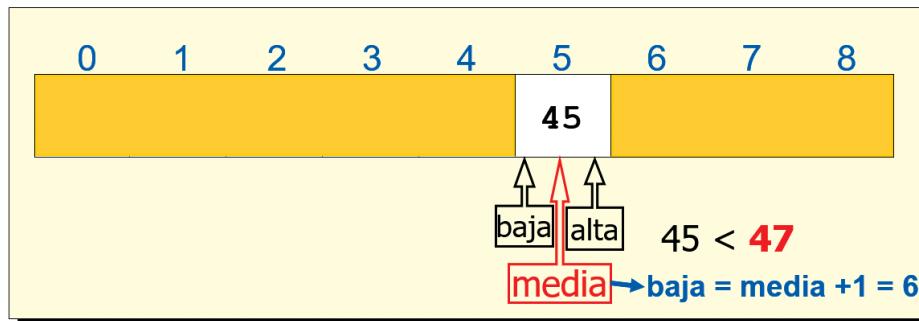


Figura 15. Ejemplo de búsqueda binaria fallida. Tercera pasada

	baja	alta	media	búsqueda(47)
#1	0	8	4	
#2	5	8	6	
#3	5	5	5	
#4	6	5		$media = \left\lfloor \frac{baja + alta}{2} \right\rfloor$

0 1 2 3 4 5 6 7 8

Búsqueda fallida

alta baja

no hay mas elementos a buscar ← $baja > alta$

Figura 16. Ejemplo de búsqueda binaria fallida. Cuarta pasada

EJEMPLO:

Realizar el seudocódigo del método de búsqueda binaria y su implementación en Java.

```
busquedaBinaria(lista[], valor)
    alta,baja,media:ENTERO
    baja = 0
    alta = lista.longitud-1
    mientras(baja<=alta)
        media = (alta+baja)/2
        si lista[media]==valor
            retornar media
        si no si valor<lista[media]
            alta = media-1
```

```
        si no
            baja = media+1
        finSi
    finSi
    finCiclo
    retornar -1
FIN

public static int busquedaBinaria(int[] lista, int valor) {
    int alta, baja, media;
    baja = 0;
    alta = lista.length-1;
    while(baja<=alta){
        media = (alta+baja)/2;
        if(lista[media]==valor)
            return media;
        else if(valor<lista[media])
            alta = media-1;
        else
            baja = media+1;
    }
    return -1;
}
```

1.19. Ordenando un arreglo

El ordenamiento es una tarea común en la computación.

Cuando tenemos una colección de datos, muchas aplicaciones necesitan ordenar los datos en cierto orden. Por ejemplo, ordenar la información de una persona en forma ascendente por edad, ordenar los archivos del disco duro alfabéticamente, etc.

EJEMPLO:

- Ordenar mails en su correo por fecha, por remitente, etc.
- Ordenar canciones por título, por autor, etc.
- Ordenar estudiantes por su CUI, por su promedio, alfabéticamente, etc.

Definición del problema:

Dado un arreglo de n valores, ordenar sus valores con algún criterio de orden (ascendente o descendente).

Existen varios algoritmos de ordenamiento, veremos algunos a continuación.

1.20. Ordenamiento de Burbuja

Asumimos que se desea un ordenamiento ascendente.

Se realizan varias pasadas, en cada pasada se garantiza que el valor más pesado se hunda al fondo.

Para cada pasada se van comparando cada dos elementos, intercambiando

cuando el primero es mayor que el segundo.

Dados n elementos se necesitarán $n-1$ pasadas para garantizar que el arreglo esté ordenado.

Considerar que en cada pasada, a partir de la segunda, ya no se necesitará comparar con los elementos que ya están ordenados.

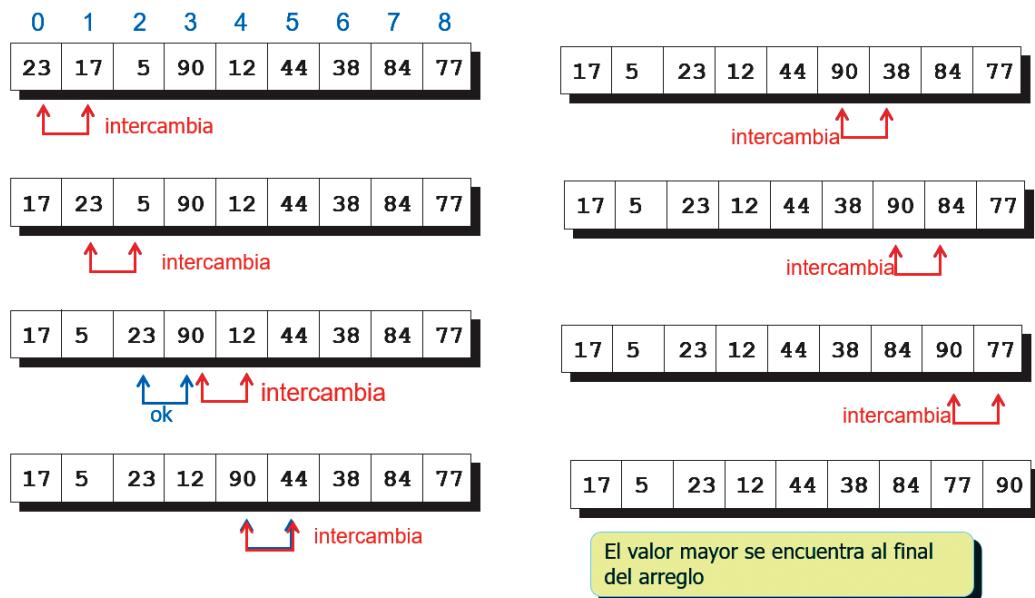


Figura 17. Ejemplo de ordenamiento de burbuja: primera pasada

EJEMPLO:

Realizar la implementación del método de ordenamiento de burbuja en Java para un arreglo de enteros.

```
public static void ordenarBurbuja(int[] lista) {
    int temp;
    for (int i=1; i<lista.length; i++)
        for(int j=0;j<lista.length-i;j++)
            if(lista[j]>lista[j+1]){
                temp = lista[i];
                lista[i] = lista[j];
                lista[j] = temp;
            }
}
```

1.21. Ordenamiento por Selección

Existen varios algoritmos de ordenamiento con diferentes grados de complejidad y eficiencia.

En el ordenamiento por selección se realizan varias pasadas, en cada pasada encuentra el valor más pequeño en el arreglo desde la posición i hasta el final y se intercambia el valor encontrado con $\text{arreglo}[i]$.

Para la primera pasada, intercambiar el elemento de la primera posición con el elemento menor de todo el arreglo. Ahora el elemento menor está en la primera

posición.

Dados n elementos se necesitarán $n-1$ pasadas para garantizar que el arreglo esté ordenado.

Considerar que en cada pasada, a partir de la segunda, ya no se necesitará comparar con los elementos que ya están ordenados.

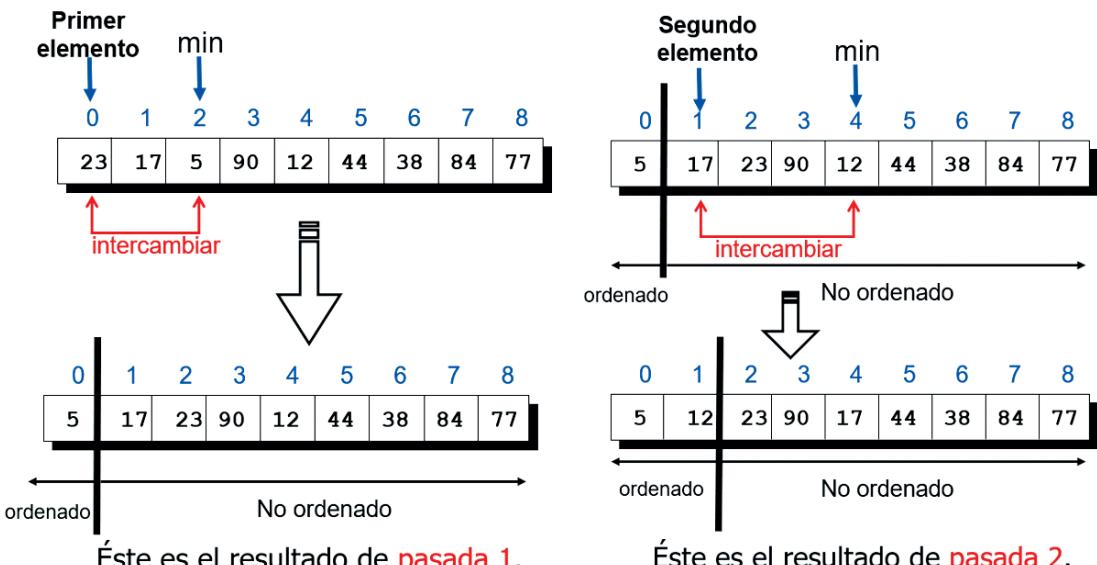


Figura 18. Ejemplo de ordenamiento por selección: arreglo original y ordenado (para 2 pasadas)

EJEMPLO:

Realizar la implementación del método de ordenamiento por selección en Java para un arreglo de enteros.

```
public static void ordenarSeleccion(int[] lista) {
    int j,temp;
    for (int i=0; i<lista.length-1; i++){
        j = indiceProximoMenor(lista, i);
        temp = lista[i];
        lista[i] = lista[j];
        lista[j] = temp;
    }
}
private static int indiceProximoMenor(int[] lista,int index){
    int minIndex = index;
    for (int i=index+1; i<lista.length; i++)
        if (lista[i] < lista[minIndex])
            minIndex = i;
    return minIndex;
}
```

1.22. Otros métodos de ordenamiento

Existen además, algoritmos de ordenamiento tales como:

- Por Inserción
- QuickSort
- MergeSort
- HeapSort

Se pueden ver videos demostrativos de los métodos de ordenamiento en:

- <http://www.sorting-algorithms.com>
- <https://www.youtube.com/watch?v=EdUWyka7kpI>
- <https://www.youtube.com/watch?v=NiyEqLZmngY>
- <http://cs.smith.edu/~thiebaut/java/sort/demo.html>
- <https://www.youtube.com/watch?v=aQiWF4E8flQ>
- <https://www.youtube.com/watch?v=HsZ-YRQM8sE>
- <http://youtube.com/watch?v=EreoMaOBTzE>
- <https://www.youtube.com/watch?v=B7hVxCmfPtM>

1.23. Arreglos bidimensionales

Son una generalización de los arreglos unidimensionales y son útiles para representar información tabular, información que se puede representar en una tabla.

Son arreglos con filas (horizontales) y columnas (verticales), tienen 2 índices y 2 dimensiones.

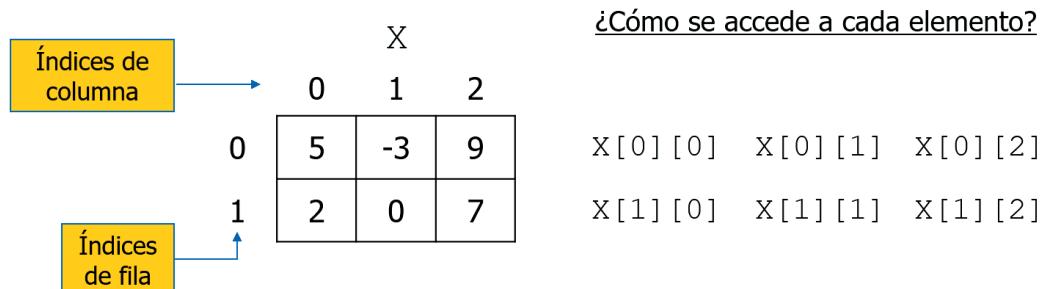


Figura 19. Estructura de un arreglo bidimensional y cómo se accede a cada elemento

Si se tienen datos que son organizados en un formato de tabla (filas y columnas), considerar usar arreglos bidimensionales.

EJEMPLO:

Queremos almacenar las tablas de multiplicar de los números enteros desde 1 hasta 9, con multiplicadores desde 1 hasta 9. Un arreglo bidimensional sería la solución.

	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9
2	2	4	6	8	10	12	14	16	18
3	3	6	9	12	15	18	21	24	27
4	4	8	12	16	20	24	28	32	36
5	5	10	15	20	25	30	35	40	45
6	6	12	18	24	30	36	42	48	54
7	7	14	21	28	35	42	49	56	63
8	8	16	24	32	40	48	56	64	72
9	9	18	27	36	45	54	63	72	81

Figura 20. Tablas de multiplicar de varios números con multiplicadores del 1 al 9

1.24. Declaración y creación de arreglos bidimensionales

Declaración:

```
int[ ][ ] lista;
```

Creación:

```
lista = new int[2][5]; // 2 filas y 5 columnas
```

Declaración y creación en una sola línea de código:

```
int[ ][ ] lista = new int[2][5];
```

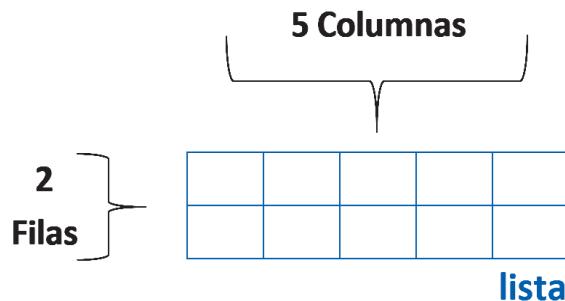


Figura 21. Arreglo bidimensional llamado "lista" de 2 filas y 5 columnas

EJEMPLO:

Crear un arreglo pagos de 4 filas y 5 columnas de elementos dobles. Luego ingresar valores por teclado para todos sus elementos.

```
double[][] pagos;
pagos = new double[4][5];

for (int i = 0; i < pagos.length; i++)
    for (int j = 0; j < pagos[i].length; j++)
        pagos[i][j] = scan.nextInt();
```

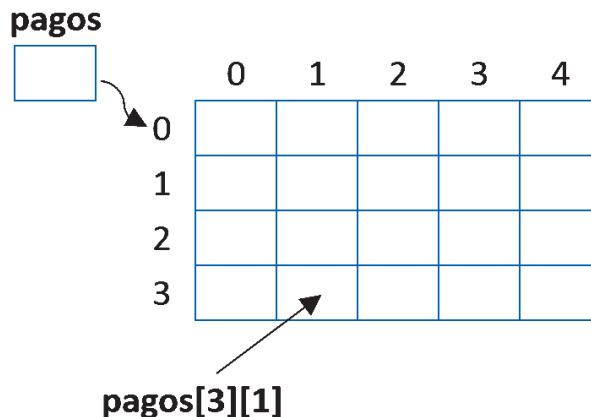


Figura 22. Arreglo bidimensional de 4x5

1.25. Otra forma de creación de arreglos bidimensionales

También se puede crear e inicializar un arreglo bidimensional utilizando una lista inicializadora.

EJEMPLO:

```
int[][] b = { { 1, 2 }, { 3, 4 } };
int[][] c = { { 1 }, { 3, 4 } };
```

Observar que los arreglos bidimensionales en Java son arreglos unidimensionales de arreglos unidimensionales, por lo que cada fila puede tener diferente número de elementos. Incluso podemos crear el arreglo sólo indicando la cantidad de filas, no de columnas y después ir creando cada fila independientemente.

EJEMPLO:

```
int[][] lista = new int[2][ ];
lista[0] = new int[5];
lista[1] = new int[3];
```

Aquí creamos un arreglo bidimensional de 2 filas, pero con la primera fila con 5 elementos y la segunda con 3. No necesariamente los arreglos bidimensionales tienen que ser rectangulares.

EJEMPLO:

Describir el arreglo que crea el siguiente código:

```
double[][] triangular = new double[4][ ];
for (int i = 0; i < 4; i++)
    triangular[i] = new double [i + 1];
```

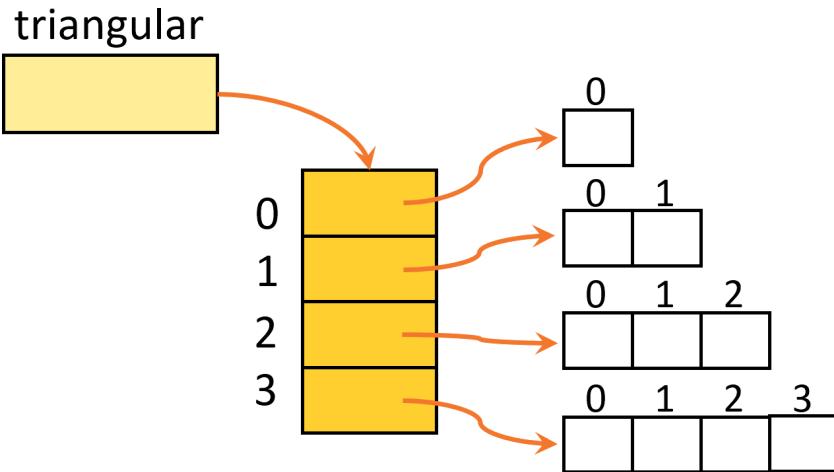


Figura 23. Arreglo bidimensional en la memoria RAM

Observar que lo que se crea es un arreglo unidimensional de arreglos unidimensionales.

EJEMPLO:

Crear el método `uno` que pone a 1 todos los elementos del arreglo `a`.

```
public void uno(int[][] a){
    for (int r = 0; r < a.length; r++)
        for (int c = 0; c < a[r].length; c++)
            a[r][c] = 1;
}
```

EJEMPLO:

Crear el método `sumar` que retorne un nuevo arreglo con los valores de la suma de otros 2 arreglos bidimensionales. Considerar $c_{i,j} = a_{i,j} + b_{i,j}$.

$$\begin{pmatrix} 5 & 2 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix} + \begin{pmatrix} -1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & 2 & -1 \end{pmatrix} = \begin{pmatrix} 4 & 2 & 0 \\ 2 & 2 & 1 \\ 1 & 2 & 0 \end{pmatrix}$$

Figura 24. Suma de arreglos bidimensionales

```
public static double[][] suma(double[][] a, double[][] b) {
    int m = a.length;
    int n = a[0].length;
    double[][] c = new double[m][n];

    for (int i = 0; i < m; ++i)
        for (int j = 0; j < n; ++j)
            c[i][j] = a[i][j] + b[i][j];
    return c;
}
```

EJERCICIO 4:

Crear el método multiplicar que retorne un nuevo arreglo con los valores de la multiplicación de 2 arreglos bidimensionales. Considerar que no siempre se pueden multiplicar 2 arreglos bidimensionales, ya que se debe cumplir cierta condición.

$$c_{ij} = (a_{i,1} \times b_{1,j}) + (a_{i,2} \times b_{2,j}) + \dots + (a_{i,n} \times b_{n,j})$$

$$\begin{pmatrix} 1 & 2 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} -1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & 2 & -1 \end{pmatrix} = \begin{pmatrix} 3 & 2 & 0 \\ 2 & 3 & -1 \\ -1 & 2 & -1 \end{pmatrix}$$

Figura 25. Producto de arreglos bidimensionales

EJERCICIO 5:

Crear el programa `Tiempos de Vuelo` que permita consultar los tiempos de vuelo de una ciudad a otra en el Perú.

Contiene un método `mostrarTablaTiemposVuelo` que imprime toda la tabla.

Contiene un método `preguntarPorTiempoVuelo` que pregunta al usuario por la ciudad de partida y la ciudad de destino, e imprime el tiempo de vuelo correspondiente.

Considerar el siguiente ejemplo de salida del programa:

Tiempos de Vuelo Perú:

	AQP	JUL	CUZ	TCQ	LIM
AQP	0	22	30	42	57
JUL	23	0	15	25	58
CUZ	31	17	0	24	30
TCQ	45	27	25	0	95
LIM	59	58	30	97	0

1 = AQP

2 = JUL

3 = CUZ

4 = TCQ

5 = LIM

Ingrrese el número de ciudad de partida: 1

Ingrrese el número de ciudad de llegada: 5

Tiempo de vuelo: 57 minutos.

Figura 26. Salida del programa `Tiempos de Vuelo`

EJERCICIOS PROPUESTOS

1. Simular n lanzamientos de un dado. Mostrar las frecuencias de cada posible valor obtenido del lanzamiento. Indicar cuál es el valor con mayor frecuencia.
2. En un juego de dados que consiste en lanzar 2 dados y ver su suma. Imprimir tipo tabla las frecuencias de cada suma posible y su histograma correspondiente. Confirmar que el 7 es la suma con más probabilidad y que 2 y 12 son los de menor probabilidad. Hacer experimento con n lanzamientos.
3. Simular las notas [0..20] de una clase de n estudiantes, calcular la frecuencia, indicar la nota con mayor y menor frecuencia, y que muestre tipo histograma los resultados.
4. Leer enteros entre 1 y 5 (calificaciones) en un arreglo de 10 posiciones. Que el programa muestre el arreglo como si fuera un arreglo de tamaño 5×2 . Además, que muestre el histograma de frecuencias.
5. Implementar la búsqueda lineal para elementos de un arreglo de n elementos enteros. 2 versiones: a) hasta encontrar el valor y b) búsqueda completa. Mostrar las posiciones donde encuentra el valor o -1 si no lo encuentra.
6. Simular un lanzamiento de n dados para m concursantes, indicar el ganador (quien obtuvo la mayor suma) y el valor del lanzamiento del dado con mayor frecuencia.
7. Implementar la suma, resta, multiplicación punto (multiplicación elemento a elemento) de 2 matrices $m \times n$.
8. Generador de matrices identidad de $n \times n$.
9. Transpuesta de una matriz $m \times n$.
10. Multiplicación de matrices $m \times n$ y $n \times p$.
11. Búsqueda lineal de un entero en un arreglo de $m \times n$ elementos enteros. Hacer versión con búsqueda completa (devolver todas las posiciones en que encontró el valor).
12. Implementar la búsqueda binaria para elementos de un arreglo de n elementos String.
13. Implementar la técnica de ordenamiento de burbuja para Strings.
14. Generar aleatoriamente las notas [0..20] de una clase de n estudiantes, imprimir considerando su orden de creación e imprimirla ordenada del mejor al peor estudiante.
15. Simular un sorteo de orden de 1 al 6 (cada número sólo se repite una vez). Programa iterativo. Por ejemplo: un orden 5 3 2 6 4 1, otro orden 1 6 2 3 4 5.
16. Usando métodos ingresar(), ordenar(), imprimir(). Hacer un programa donde se ingresen n elementos enteros a un arreglo, se impriman, se ordenen y se vuelvan a imprimir.
17. Implementar métodos de Ordenamiento por Selección y Ordenamiento por Inserción para elementos de tipo de dato String.

CAPÍTULO 2

ARRAYLIST Y HASHMAP

OBJETIVOS:

- Comprender a la colección `ArrayList` como una alternativa a los arreglos estándar
- Comprender a la colección `HashMap` como una estructura de datos alternativa
- Saber declarar, crear e inicializar `ArrayList` y `HashMap`, y referirse a sus elementos, usando sus diferentes métodos
- Aprender a utilizar los `ArrayList` unidimensionales, bidimensionales y `HashMap`
- Aplicar los `ArrayList` y `HashMap` en la resolución de problemas

2.1. Motivación

Problema: imagine que usted quiere almacenar los nombres de todos sus estudiantes, pero no sabe a priori cuántos estudiantes tiene.

Alternativas:

- Crear un arreglo suficientemente grande como para almacenar todos los posibles nombres (se podría provocar desperdicio de espacio si tiene menos estudiantes de los que consideró o podría generar una falta de espacio si tiene más estudiantes de lo previsto).
- Utilizar el potencial de Java de creación de arreglos en tiempo de ejecución (surgiría el mismo problema que en la alternativa anterior, ya que no se sabe la cantidad de estudiante a priori).
- Utilizar los `ArrayList`.

2.2. Generalidades

`ArrayList` es una colección (Collection): objeto que almacena datos y forma parte de la API de Java.

La clase `ArrayList` provee la funcionalidad básica que viene con un arreglo estándar, pero añade una funcionalidad extra.

La funcionalidad básica: un `ArrayList` almacena una colección ordenada de valores y permite el acceso a dichos valores por medio de un índice.

La funcionalidad añadida: un `ArrayList` crece y se comprime dinámicamente al insertar y borrar elementos en cualquier ubicación especificada (se redimensiona automáticamente).

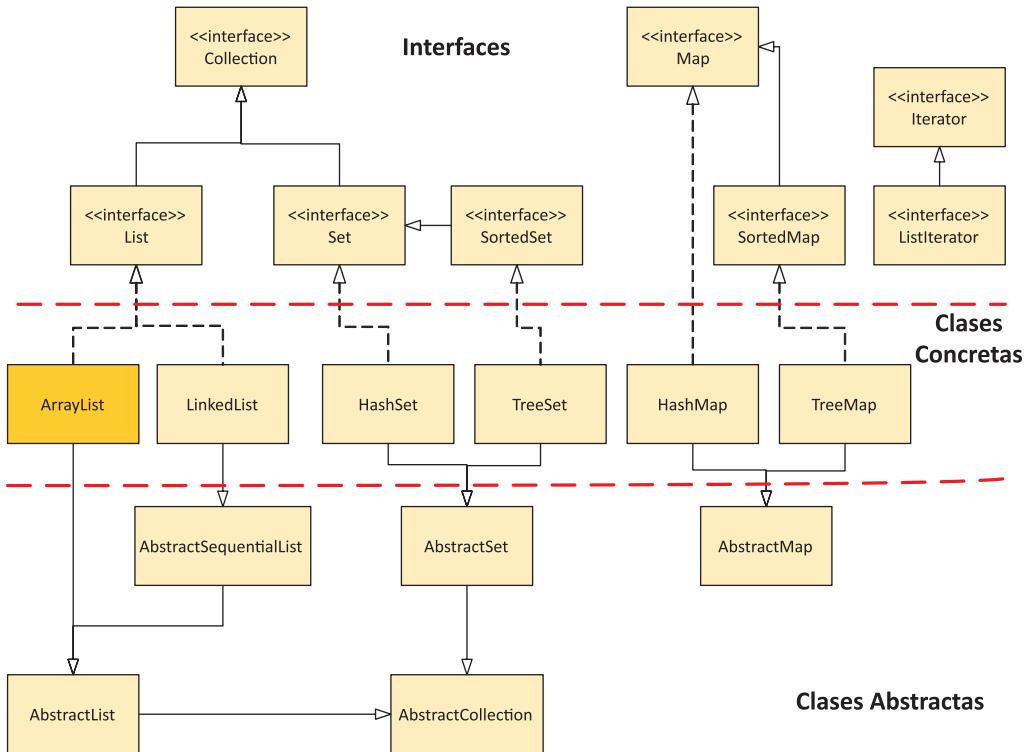


Figura 27. Ubicación del ArrayList en la API de Java

La clase `ArrayList` está definida en el paquete `java.util` de la API de Java, así debemos importar el paquete para poder usarla. Tenemos 2 opciones para esto:

```
import java.util.*;
import java.util.ArrayList;
```

Para declarar una variable de referencia y crear un objeto `ArrayList`, usar la sintaxis:

```
ArrayList<tipoElem> variableRef = new ArrayList<tipoElem>();
```

EJEMPLO:

Crear el `ArrayList` `estudiantes` que almacene los nombres de los estudiantes:

```
ArrayList<String> estudiantes = new ArrayList<String>();
```

Comparar con el siguiente arreglo estándar e indicar sus diferencias:

```
String[] estudiantes = new String[100];
```

2.3. Diferencias entre los ArrayList y los arreglos estándar

- Usa `<>` para especificar el tipo de los elementos.
- `<tipoElem>` debe ser el nombre de una clase (no de un tipo de dato primitivo).
- Al crear un `ArrayList` no se especifica el número de elementos, ya que los objetos de `ArrayList` se inician sin elementos.

2.4. Métodos de la clase ArrayList

1. Método add

Para añadir un elemento al final de un objeto `ArrayList`, usamos el método `add`:

```
variableArrayList.add(item);
```

El `item` que es añadido debe ser del mismo tipo que el especificado en la declaración y creación del `ArrayList`.

EJEMPLO:

Escribir el código que crea el siguiente objeto `ArrayList`:

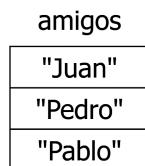


Figura 28. `ArrayList` de `String`

```
import java.util.ArrayList;
. . .
ArrayList<String> amigos = new ArrayList<String>();
amigos.add("Juan");
amigos.add("Pedro");
amigos.add("Pablo");
```

EJEMPLO:

Observar el siguiente diagrama de memoria del `ArrayList` creado y ver cómo lo que dicho `ArrayList` realmente almacena son las referencias de los objetos añadidos.

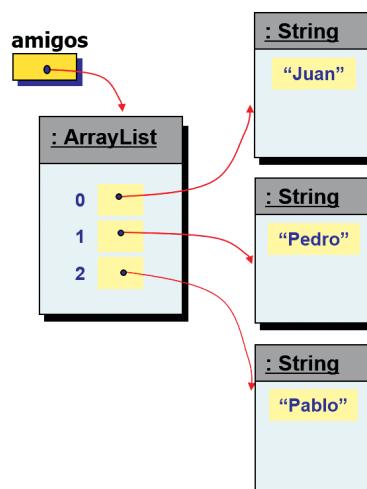


Figura 29. Diagrama de memoria del `ArrayList` "amigos"

2. Método `get`

Para acceder a un elemento específico del `ArrayList` (leer).

Cabecera del método: `public E get(int index)`

- `index`: posición dentro del `ArrayList`. Empieza en 0 como los arreglos estándares
- Si `index` se refiere a un elemento no existente, ocurrirá un error de tiempo de ejecución
- Si `index` es válido, `get` retornará el elemento que se encuentra en dicha posición
- `E` se entiende por Elemento. Representa el tipo de dato de los elementos que almacena el `ArrayList`

EJEMPLO:

Crear el `ArrayList` `estudiantes` para almacenar los nombres de los estudiantes, añadir como primer estudiante a "Juan", imprimir el nombre añadido y luego hacer que la variable `s` almacene su valor.

```
ArrayList<String> estudiantes = new ArrayList<String>();  
  
estudiantes.add("Juan");  
System.out.println(estudiantes.get(0));  
String s = estudiantes.get(0);
```

3. Método `set`

Permite asignar un valor a un elemento del `ArrayList`. Sobrescribe un elemento existente.

Cabecera del método: `public E set(int index, E elem)`

- `index`: posición dentro del `ArrayList`. Empieza en 0 como los arreglos estándares
- Si `index` se refiere a un elemento no existente, ocurrirá un error de tiempo de ejecución
- Si `index` es válido, `set` asignará el parámetro `elem` al elemento especificado, sobrescribiendo lo que haya originalmente
- `E` representa el tipo de dato de los elementos del `ArrayList`. Retorna el elemento reemplazado, el cual puede ser capturado en una variable o no.

EJEMPLO:

¿Cómo quedará el `ArrayList` después de ejecutar el siguiente código?

```
String mezcla;  
ArrayList<String> amigos = new ArrayList<String>();  
  
System.out.println(amigos);  
amigos.add("Juan");  
amigos.add("Pedro");
```

```
amigos.add("Pablo");
System.out.println(amigos);
mezcla = amigos.get(0) + amigos.get(1);
colors.set(2, mezcla);
System.out.println(amigos);
```

[]
[Juan, Pedro, Pablo]
[Juan, Pedro, JuanPedro]

4. Otros métodos

- `public int size()`
Retorna el número de elementos en la lista.
- `public void add(int index, E elem)`
Inserta el `elem` en la posición `index` indicada. Desplazando los elementos originales a posiciones mayores.
- `public void clear()`
Borra todos los elementos del `ArrayList`.
- `public int indexOf(E elem)`
Busca la primera ocurrencia del parámetro `elem` dentro del `ArrayList`. Si lo encuentra retorna la posición correspondiente dónde lo encontró, si no lo encuentra devuelve `-1`.
- `public boolean isEmpty()`
Retorna `true` si la lista no contiene elementos.
- `public E remove(int index)`
Quita el elemento ubicado en la posición indicada `index`, desplaza todos los elementos con índice mayor a posiciones menores, además retorna el elemento borrado (elemento que puede ser capturado en una variable o no).
- `public boolean contains(E elem)`
Retorna `true` si la lista contiene al parámetro `elem`.
- `public int lastIndexOf (E elem)`
Busca la última ocurrencia del parámetro `elem` dentro del `ArrayList`. Si lo encuentra retorna la posición correspondiente dónde lo encontró, si no lo encuentra devuelve `-1`.

EJEMPLO:

Realizar un programa utilizando `ArrayList` donde se añadan 5 nombres de personas, luego que aleatoriamente se elimine alguno de ellos, se imprima el nombre eliminado y finalmente se imprima los nombres que quedan.

```

import java.util.ArrayList;

public class Ejemplo12fp {

    public static void main(String[] args) {
        int indicePerdedor;
        String elPerdedor;
        ArrayList<String> tribu = new ArrayList<String>();
        tribu.add("Ricardo");
        tribu.add("Jose");
        tribu.add("Adrian");
        tribu.add("Ana");
        tribu.add("Carlitos");
        indicePerdedor = (int) (Math.random() * 5);
        elPerdedor = tribu.remove(indicePerdedor);
        System.out.println("Lo siento, " + elPerdedor +
            ". La tribu ya eligio, quedas eliminado.");
        System.out.println("Aun quedan: " + tribu);
    }
}

```

2.5. Otras consideraciones sobre los ArrayList

Si se intenta imprimir o concatenar un ArrayList, el ArrayList retorna una lista separada por comas de los elementos del ArrayList y enmarcados por corchetes [].

La última línea del ejemplo anterior imprimiría:

Aun quedan: [Ricardo, Jose, Ana, Carlitos]

2.6. Usando tipos de datos primitivos con ArrayList

Un ArrayList sólo puede almacenar objetos, no puede almacenar datos de tipo primitivo.

En el programa anterior, tribu es un ArrayList de String y los String son objetos (o de tipo de dato por referencia).

Para almacenar tipos de datos primitivos, se deben usar las clases wrapper (envoltorios) correspondientes.

Así, ArrayList<int> ¡no es válido!, pero ArrayList<Integer> si lo es.

Tipo Primitivo	Wrapper Class
int	Integer
double	Double
char	Character
boolean	Boolean

Tabla 2. Clases wrapper (envoltorio) de Java

EJEMPLO:

Crear un `ArrayList` de enteros. Observar que no se puede hacer directamente con `int`, sino se debe utilizar su clase wrapper correspondiente `Integer`.

```
ArrayList<Integer> lista = new ArrayList<Integer>();
```

EJEMPLO:

Programa que almacene 4 enteros en un `ArrayList` y que muestre todo su contenido y la suma de los elementos.

```
import java.util.ArrayList;

public class Ejemplo13fp {

    public static void main(String[] args) {
        int suma = 0;
        ArrayList<Integer> lista = new ArrayList<Integer>();

        lista.add(13);
        lista.add(50);
        lista.add(15);
        lista.add(9);

        for (int n : lista)
            suma += n;
        System.out.println("lista = " + lista);
        System.out.println("suma = " + suma);
    }
}
```

Salida:

```
lista = [13, 50, 15, 9]
suma = 87
```

2.7. **Sentencia for-each**

Es una sentencia alternativa al `for`, nos permite recorrer completamente cualquier colección de datos tal como un `ArrayList`, un arreglo estándar, etc.

SINTAXIS:

```
for (<tipoElemento> <nombreElemento> : <ArrayList/Arreglo>){
    <sentencias>;
}
```

EJEMPLO:

Evaluar el siguiente bucle `for-each`:

```
for(String nombre:tribu)
    System.out.println(nombre);
```

Se lee: "para cada nombre dentro de tribu imprimir nombre".

¿Su equivalente en for?

```
for(int i=0;i<tribu.size();i++)
    System.out.println(tribu.get(i));
```

La implementación usando for-each puede ser preferida ya que es más simple.

EJEMPLO:

Usando for – each para recorrer un arreglo estándar:

```
int[] primos = {2, 3, 5, 7, 11};
for (int p : primos)
    System.out.println(p);
```

EJERCICIO 6:

Crear un ArrayList de String, almacenar 3 elementos y mostrar la suma de las longitudes (cantidad de caracteres) de dichos elementos.

2.8. Métodos con ArrayList

1) Método que recibe un ArrayList

```
<modificadores> <Tipo> nombreMetodo(ArrayList<Tipo> nombre)
```

EJEMPLO:

```
public static void imprimir(ArrayList<String> estudiantes) {
    for(int i=0;i<estudiantes.size();i++)
        System.out.println(estudiantes.get(i));
}
```

2) Método que retorna un ArrayList

```
<modificadores> ArrayList<Tipo> nombreMetodo(parámetros)
```

EJEMPLO:

```
public static ArrayList<Tipo> nombreMetodo(parámetros) {
    ...
    return x;    //x es un ArrayList del Tipo especificado
}
```

2.9. ArrayList Bidimensionales

Al igual que los arreglos estándar, los ArrayList bidimensionales son sólo una generalización de los ArrayList unidimensionales.

Cuando hablamos de un ArrayList bidimensional en realidad lo que se tiene es un ArrayList de ArrayList.

EJEMPLO:

Dado un `ArrayList` bidimensional `obj`, es muy común utilizar el encadenamiento de métodos, así se podría hacer lo siguiente:

```
obj.met1().met2();
```

Primero se llama al método `met1` y al resultado devuelto se le aplica `met2`.

La forma de declarar y crear `ArrayList` bidimensionales y los métodos principales:

- 1) Declaración y creación

```
ArrayList<ArrayList<Integer>> arr = new ArrayList<ArrayList<Integer>>();
```

- 2) Insertar una nueva fila

```
arr.add(new ArrayList<Integer>());
```

- 3) Añadir un elemento a una fila específica

```
arr.get(fila).add(valor);
```

- 4) Tamaño de filas y columnas

`array.size()` cantidad de filas

`array.get(fila).size()` cantidad de columnas (elementos de fila)

- 5) Escribir datos en posición específica. Debe haber algún elemento en dicha posición `fila, col`

```
array.get(fila).set(col, 5);
```

- 6) Leer datos de posición específica

```
System.out.print(array.get(fila).get(col)+" ");
```

EJEMPLO:

Crear un `ArrayList` bidimensional de 5 filas y 3 columnas, añadir valores enteros que sean la suma de su número de fila con su número de columna. Cambiar el último valor de la segunda fila a 69. Finalmente recorrer el `ArrayList` imprimiendo en forma matricial sus elementos.

```
import java.util.*;  
  
public class Ejemplo14fp {  
  
    public static void main(String[] args) {  
        ArrayList<ArrayList<Integer>> array = new ArrayList<ArrayList<Integer>>();  
  
        for(int i=0;i<5;i++){  
            array.add(new ArrayList<Integer>());  
            for(int j=0;j<3;j++)  
                array.get(i).add(i+j);  
        }  
        array.get(1).set(2, 69); //funciona, pero cuando existe el elemento
```

```
for(int i=0;i<array.size();i++){ //para cada fila
    System.out.print("Fila " + i + ": ");
    for(int j=0;j<array.get(i).size();j++) //recorre columnas
        System.out.print(array.get(i).get(j) + " ");
    System.out.println();
}
}
```

2.10. ArrayList vs arreglos estándar

Beneficios de un <code>ArrayList</code> sobre un arreglo estándar	Beneficios de un arreglo estándar sobre un <code>ArrayList</code>
Fácil de aumentar el tamaño de un <code>ArrayList</code> – sólo usar <code>add</code> .	Un arreglo usa <code>[]</code> para acceder a sus elementos (es más fácil que usar los métodos <code>get</code> y <code>set</code>).
Es fácil para un programador insertar o eliminar un elemento en un <code>ArrayList</code> – sólo usar <code>add</code> o <code>remove</code> y especificar el índice del elemento.	Un arreglo es más eficiente cuando se almacenan datos de tipos primitivos.

Tabla 3. ArrayList versus arreglos estándar

2.11. HashMap

Es un `Map` (Mapa), una estructura de datos definida en la API de Java.

Almacena los datos como pares <clave, valor> y tiene un cierto parecido a `ArrayList`.

Mientras `ArrayList` y `LinkedList` son implementaciones de `List`, `HashMap` es una implementación de `Map`.

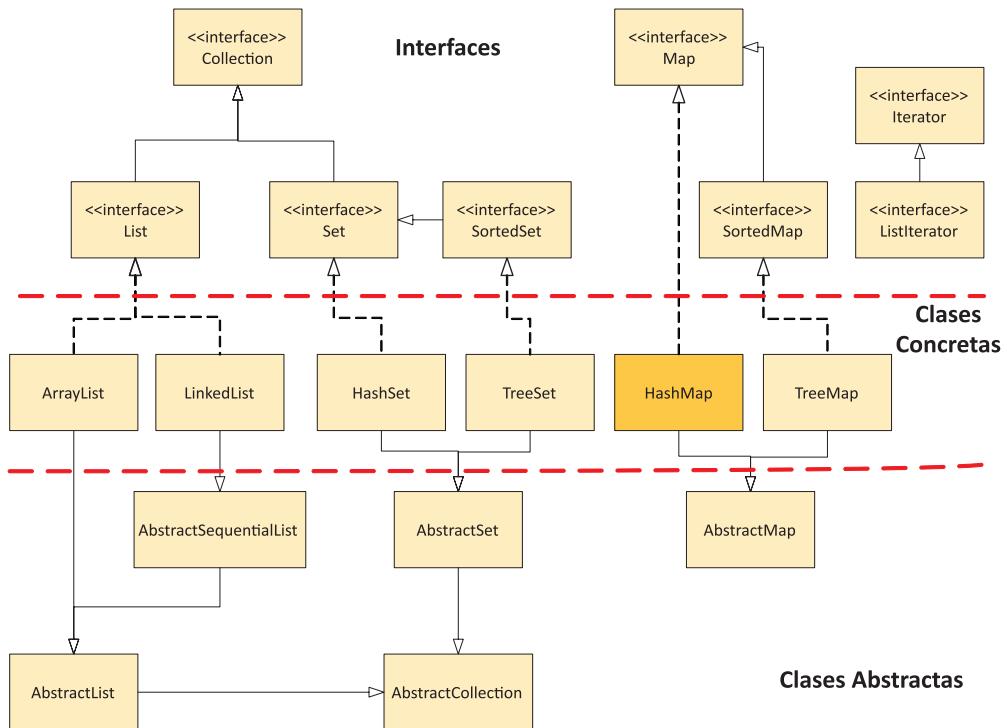


Figura 30. Ubicación del HashMap en la API de Java

Un HashMap está compuesto por entradas, cada una de las cuales es un par:

<clave, valor> o <key, value>

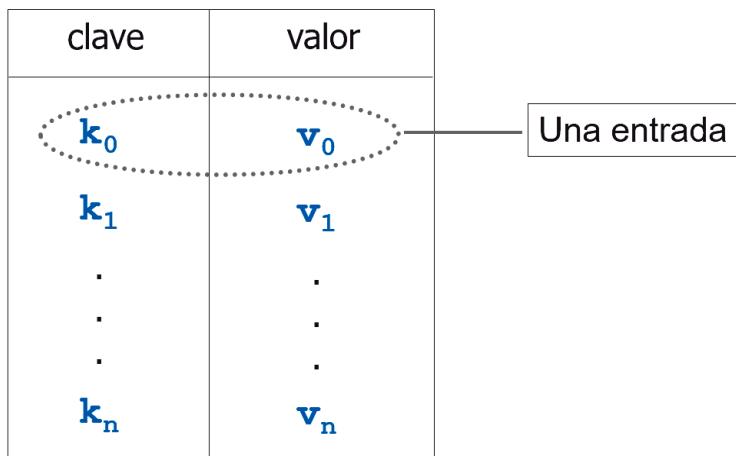


Figura 31. Estructura de HashMap de n entradas

Cada componente usado al definir un HashMap debe ser una clase y cada entrada debe ser un par de objetos correspondientes a dichas clases usadas.

Tiene la particularidad que la clave es única, así, no puede haber 2 entradas con la misma clave. Si añadimos una entrada con una clave existente, el valor se sobrescribe.

Para almacenar tipos de dato primitivos usar sus respectivas clases wrapper.

Son como arreglos, pero con la posibilidad de tener índices no enteros y su principal ventaja es la performance para encontrar la entrada dada la clave.

SINTAXIS:

Se debe importar `java.util.*`

Se debe declarar y crear.

```
HashMap<tipoClave, tipoValor> nombre;
nombre = new HashMap<tipoClave, tipoValor>();
```

O de forma compacta:

```
HashMap<tipoClave, tipoValor> nombre= new HashMap<tipoClave, tipoValor>();
```

EJEMPLO:

Crear un `HashMap` que tenga como componentes, tanto en clave como valor, a `Strings` que representen los cursos del primer semestre de Ingeniería de Sistemas.

```
HashMap<String, String> cursos = new HashMap<String, String>();
```

clave	valor
<code>"IS001"</code>	<code>"Fundamentos de Programación 1"</code>
<code>"IS002"</code>	<code>"Introducción a la Computación"</code>
<code>"IS003"</code>	<code>"Estructuras Discretas 1"</code>
...	...

Figura 32. `HashMap` de `String, String`

2.12. Principales métodos de `HashMap`

1) Método `put`

Añade la entrada `(key, value)` en el `HashMap`.

```
Object put( Object clave, Object valor )
```

2) Método `get`

Retorna el valor correspondiente a la clave dada.

```
Object get( Object clave )
```

3) Método `remove`

Quita del mapa la entrada cuya clave se envía.

```
Object remove( Object clave )
```

4) Método `containsKey`

Retorna `true` si el mapa contiene una entrada con la clave dada.

```
boolean containsKey( Object clave )
```

5) Método `containsValue`

Retorna `true` si el mapa contiene una entrada con el valor dado.

```
boolean containsValue( Object valor )
```

6) Método `size`

Retorna el número de elementos del `HashMap`.

```
int size( )
```

- 7) Método `keySet`
Retorna el conjunto de las claves.
`Set<K> keySet()`
- 8) Método `values`
Retorna una colección de valores.
`Collection<V> values()`
- 9) Método `toString`
Retorna descripción en String de todos los objetos almacenados en el `HashMap`.
`String toString()`
`{IS0001=Fundamentos de Programacion 1, IS0002=Introduccion a la Computacion}`
- 10) Método `clear`
Limpia el `HashMap`, lo vuelve vacío.
`void clear()`
- 11) Método `isEmpty`
Retorna `true` si el `HashMap` está vacío.
`boolean isEmpty()`
- 12) Método `entrySet`
Retorna una vista de Conjunto (Set).
A cada elemento, después se le puede aplicar `getKey` y `getValue` para conseguir la clave y el valor respectivamente
`Set<Map.Entry<K,V>> entrySet()`

EJEMPLO:

Crear un equipo de fútbol con 3 jugadores con sus datos: número y nombre, luego imprimir sus datos completos. Usando `HashMap`

```
import java.util.*;  
  
public class Ejemplo15fp {  
  
    public static void main(String[] args) {  
  
        HashMap<Integer, String> equipo=new HashMap<Integer, String>();  
  
        equipo.put(1,"Gallese");  
        equipo.put(2,"Tapia");  
        equipo.put(3,"Flores");  
  
        for(Integer key:equipo.keySet())  
            System.out.println(key+": "+equipo.get(key));  
    }  
}
```

EJEMPLO:

Crear un `HashMap` de 3 cursos: (código, nombre), mostrar todos, eliminar un curso, mostrar todos los que quedan, mostrar el nombre dado un código, verificar si contiene cierta entrada dado un código. Recorrer todo el `HashMap` mostrando sus datos con el formato: `IS001 : Introducción a la Computación`

Borrar todo y mostrar sus elementos.

```
import java.util.*;  
  
public class Ejemplo16fp {  
  
    public static void main(String[] args) {  
        HashMap<String, String> cursos = new HashMap<String, String>();  
  
        cursos.put("IS0001", "Fundamentos de Programacion 1");  
        cursos.put("IS0002", "Introduccion a la Computacion");  
        cursos.put("IS0003", "Estructuras Discretas 1");  
        System.out.println(cursos);  
        cursos.remove("IS0002");  
        System.out.println(cursos);  
  
        String nombre = cursos.get("IS0001");  
        System.out.println(nombre);  
  
        boolean resultado = cursos.containsKey("IS0003");  
        System.out.println(resultado);  
  
        for (String key:cursos.keySet())  
            System.out.println(key+": "+cursos.get(key));  
  
        for (Map.Entry<String, String> entrada:cursos.entrySet())  
            System.out.println(entrada.getKey()+"："+entrada.getValue());  
  
        cursos.clear();  
        System.out.println(cursos);  
    }  
}
```

EJEMPLO:

Crear un generador de códigos de ciudades.

Se almacenan códigos y nombres de ciudades hasta que se desee, el código (`key`) se autogenera a partir de los 2 primeros caracteres del nombre de la ciudad y en mayúsculas.

Luego se consulta, dado el código, qué ciudad le corresponde. Hacerlo iterativo.

```
import java.util.*;  
  
public class Ejemplo17fp {  
  
    public static void main(String[] args) {  
        HashMap<String, String> postalMap = new HashMap<String, String>();  
        String nombre;  
        Scanner scan = new Scanner(System.in);  
  
        while(true) {  
            System.out.println("Ingresar nombre de ciudad? (ENTER si no hay):");  
            nombre = scan.nextLine();  
  
            if(nombre.isEmpty()) break;  
  
            String key = nombre.substring(0,2).toUpperCase();  
            String value = nombre;  
            System.out.println("ciudad = " + value + " y clave = " + key);  
            postalMap.put(key, value);  
        }  
  
        while(true) {  
            System.out.println("Ingresar el codigo? (ENTER si quieres salir):");  
            String codigo = scan.nextLine().toUpperCase();  
            if(codigo.isEmpty()) break;  
            System.out.println(codigo.length());  
  
            String ciudad = postalMap.get(codigo);  
            if(ciudad==null)  
                System.out.println("No hay tal abreviacion, intenta de nuevo");  
            else  
                System.out.println("el codigo "+codigo+" es ciudad: "+ciudad);  
        }  
    }  
}
```

EJERCICIOS PROPUESTOS

ARRAYLIST

1. Inicializar aleatoriamente un `ArrayList` de n elementos enteros, imprimir el promedio, el mayor, el menor y que muestre sólo los elementos con valor par.
2. En un juego de dados que consiste en lanzar 2 dados y ver su suma. Confirmar que el 7 es la suma con más probabilidad. Hacer un experimento que confirme la afirmación anterior con n lanzamientos. Que muestre el número de mayor frecuencia y de menor.
3. Simular las notas [0..20] de una clase de n estudiantes, calcular la frecuencia y que muestre tipo histograma los resultados.
4. Simular un sorteo de orden de 1 al 6 (cada valor sólo se repite una vez), que sea un programa iterativo.
5. Simular las notas [0..20] de una clase de n estudiantes, imprimir en orden de creación y ordenadas.
6. Implementar la suma, resta, multiplicaciónPunto de 2 `ArrayList` $m \times n$. Inicializar con valores aleatorios entre 1 y 6. Imprimir en forma matricial .
7. Generador de `ArrayList` identidad de cualquier dimensión.
8. Transpuesta de un `ArrayList` de cualquier dimensión.
9. Multiplicación de `ArrayList` de cualquier dimensión.
10. Inversa de un `ArrayList` bidimensional cualquiera.

HASHMAP

11. Crear un generador de códigos de cursos.

Se almacenan códigos y nombres de cursos hasta que se desee, el código (`key`) se autogenera a partir de los 2 primeros caracteres del nombre del curso y en mayúsculas.

Luego se consulta, dado el código, qué curso le corresponde. Hacerlo iterativo.

Considerar que puede darse el caso que 2 ó más cursos comiencen con los 2 mismos caracteres y el programa debe dar alguna solución.

12. Definir `Iterator` y resolver los 3 ejemplos desarrollados usándolo.

CAPÍTULO 3

FUNDAMENTOS DE LA ORIENTACIÓN A OBJETOS

OBJETIVOS:

- Comprender los conceptos de clase, objeto, atributo y método
- Aprender a crear clases y objetos propios
- Valorar las ventajas de la orientación a objetos
- Representar clases con UML
- Distinguir los niveles de acceso privado y público
- Comprender los métodos set, get (mutadores y accesores), métodos constructores, sobrecarga de métodos, método `toString` y métodos booleanos
- Aprender los mecanismos de comunicación con los métodos: argumentos, parámetros y tipo de dato de retorno
- Utilizar métodos que retornen un objeto
- Comprender la persistencia de los atributos, variables locales y parámetros
- Crear miembros de instancia y de clase
- Solucionar problemas aplicando el paradigma Orientado a Objetos

3.1. Antecedentes

En el pasado, el paradigma de programación predominante fue la "Programación Procedimental", en el que el énfasis estuvo en los procedimientos (tareas, funciones o métodos) que constituyan el programa. Se construían los programas alrededor de los procedimientos clave.

Hoy el paradigma predominante es el de la "Programación Orientada a Objetos" (POO). En este paradigma, en lugar de pensar primero en los procedimientos, debemos pensar en las "cosas" que son parte del problema. Estas cosas son llamadas **objetos**.

Nuestro mundo está compuesto de diferentes objetos, así todo lo que vemos e incluso lo abstracto, podemos definirlo como objeto.

Cada objeto es creado basado en una clase. Así por ejemplo, si tenemos un molde de plástico para hacer galletas, usando dicho molde podemos crear n galletas comestibles. El molde sería la clase y las galletas comestibles serían los objetos.

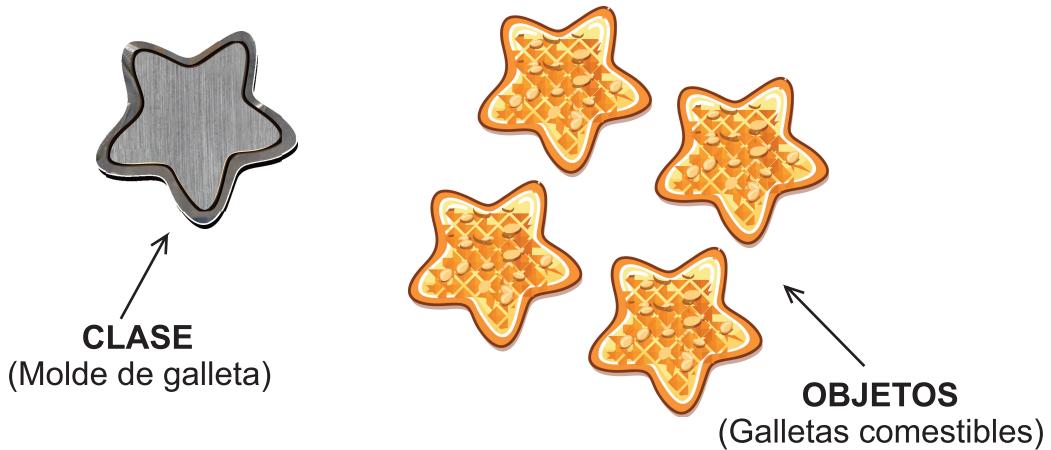


Figura 33. Clase y Objetos: molde de galleta vs galletas comestibles

Hasta ahora hemos programado creando objetos de clases estándar de la API de Java, pero en este capítulo aprenderemos a crear nuestras propias clases.

Así, para usar un objeto debemos seguir los siguientes pasos:

- 1) Declarar la variable de referencia o nombre del objeto, que sea del tipo de una clase conocida.
- 2) Crear el objeto con el operador `new` y asignarlo a la variable de referencia creada previamente. 1) y 2) pueden ir en una sola línea.
- 3) Usar o enviar mensajes al objeto.

EJEMPLO:

```
import javax.swing.*;

public class Ejemplo18fp{

    public static void main(String[ ] args){

        JFrame miVentana; 1) Declara variable de referencia o
                           nombre de tipo JFrame (API Java)
        miVentana = new JFrame( ); 2) Crea un objeto

        miVentana.setSize(300, 200); 3) Usa objeto (mensaje)
        miVentana.setTitle("Mi primer programa");
        miVentana.setVisible(true);
    }
}
```

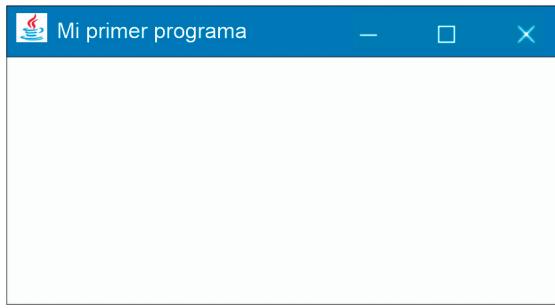


Figura 34. Ventana generada por ejecución del ejemplo

3.2. Conceptos

En la Programación Orientada a Objetos o POO, la idea básica de este tipo de programación es agrupar los atributos (datos o variables) y los métodos (funciones o procedimientos) para manejarlos en una única entidad: el **objeto**.

- Clase: Abstracción del mundo real.
- Objetos: Instancia u ocurrencia de una clase.

EJEMPLO:

Clase: Estudiante

Objetos: juan perez, jose suárez, eseEstudiante, aquelEstudiante, etc. Todos estos objetos pueden ser clasificados como "estudiantes".

En el curso anterior aprendimos a programar escribiendo todo el código en el método `main`, luego, cuando los problemas se hacían más complejos, modularizamos el código y escribimos buena parte del mismo en los **métodos**.

La orientación a objetos va más allá y modulariza creando varias clases (con sus respectivos métodos y atributos), cuyos objetos cooperan entre sí para manejar la complejidad de los problemas a resolver.

A la clase que contiene el método `main` la llamaremos **clase principal, driver o aplicación**, y sigue siendo el punto de partida de ejecución de todo el programa.

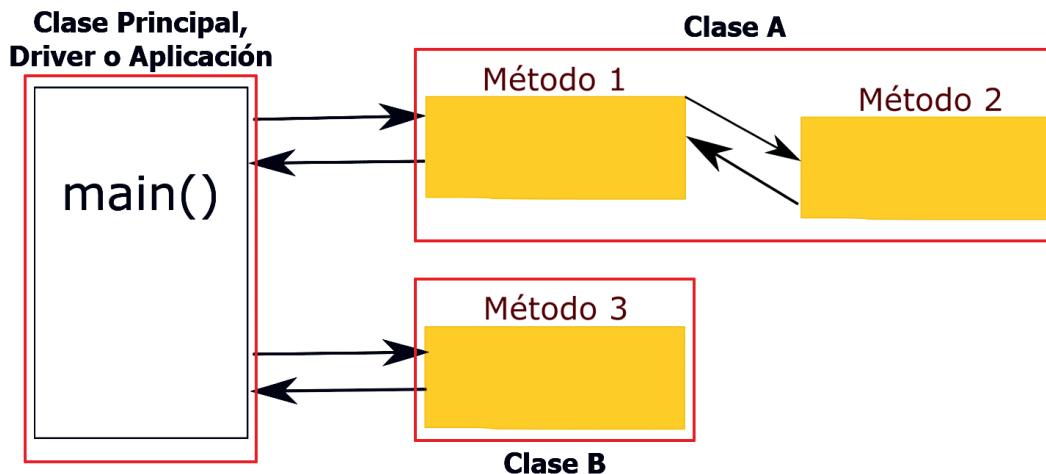


Figura 35. Esquema de un programa orientado a objetos

Existe una proximidad de los conceptos utilizados en la POO respecto a las entidades del mundo real.

EJEMPLO:

Al salir a la calle observamos una gran cantidad de objetos de tipo "Auto", los cuales los podríamos clasificar en la clase Auto.

Dicha clase Auto podría representar a todos los autos que circulan por la ciudad.

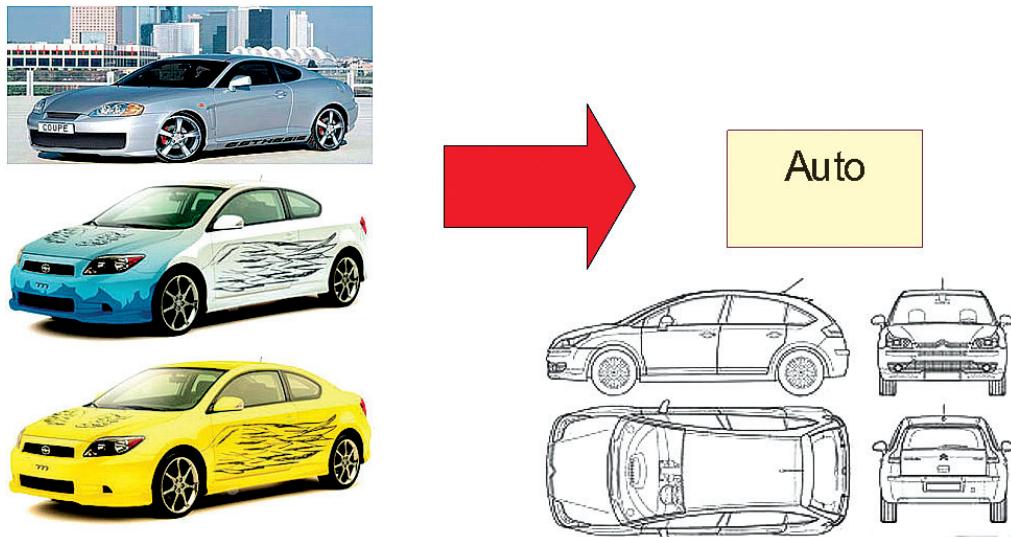


Figura 36. Objetos "Auto" y clase de donde provienen. Representación UML de Clase

Una clase integra o encapsula datos (atributos) y comportamiento (métodos) del dominio del problema.

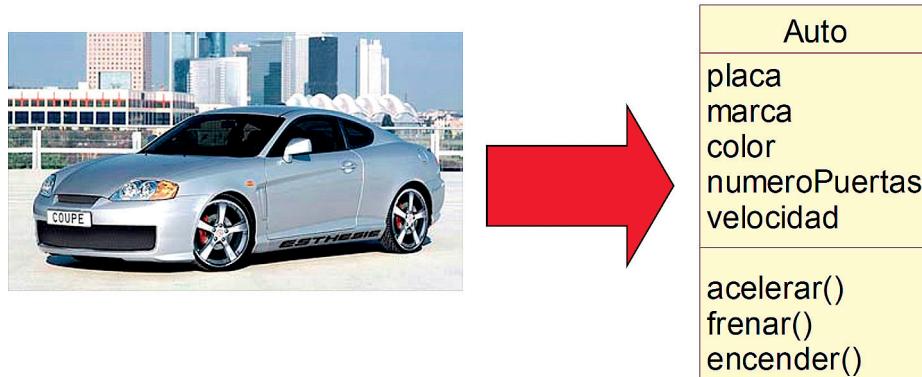


Figura 37. Representación UML de Clase, incluye atributos y métodos

- **Atributo (dato miembro o variable):** propiedad de los objetos de la clase.
- **Método (función o procedimiento):** acción, realización de una tarea. Manipula los atributos y los objetos de la clase.
- **Mensaje:** es el modo en que se comunican los objetos entre sí. En Java, un mensaje no es más que una llamada a un método de un determinado objeto.

EJEMPLO:

Clase: Estudiante
Objetos: elEstudiante, eseEstudiante, aquelEstudiante
Atributos: nombre, apellido, promedio
Métodos: cambiarPromedio(nota), cambiarNombre(nomb)
Mensaje: elEstudiante.cambiarPromedio(18)

EJERCICIO 7:

¿Qué clase y objetos tenemos? ¿atributos y métodos? ¿mensajes?. Ver siguiente figura.



Figura 38. Objetos de cierta clase

3.3. Orientación a Objetos en Java

En Java, un programa consistirá de una clase que contenga el método principal `main` (clase principal) y cero o más clases que contengan sus propios atributos (datos miembros) y métodos.

Las clases pueden ser creadas por nosotros, ser parte de la biblioteca de Java o creadas por terceros.

Es importante saber cómo tener un marco de trabajo ordenado para los programas orientados a objetos.

Si la clase será accesible por el público, se utilizará el modificador de acceso `public` y se recomienda crear un archivo `.java` con el mismo nombre que la clase que contiene, además se recomienda que el nombre de clase empiece con mayúscula.

```
//NombreClase.java          Otra Clase
public class <NombreClase> [<clase base e interfaces>] {

    private    <atributos privados>
    protected  <atributos protegidos>
    public     <atributos públicos>

    private    <métodos privados>
    protected  <métodos protegidos>
    public     <métodos públicos>
}
```

```
//Aplicacion.java      Clase Principal o Aplicación
public class Aplicacion{

    public static void main(String[] args){

        NombreClase objeto1 = new NombreClase();
        objeto1.nombreMetodo();
    }
}
```

EJEMPLO:

Crear una clase `Curso` con método `mostrarMensaje` y una clase principal donde se cree un objeto de la clase `Curso` y se invoque al método `mostrarMensaje`.

TIP: Crear el proyecto con el nombre de la clase principal `Ejemplo19fp` y para crear la clase `Curso`: en el panel de Projects expandir el proyecto `Ejemplo19fp` – expandir la carpeta `Source Packages` – en el paquete `Ejemplo19fp` click derecho y `New` – `Java Class...` – poner como nombre `Curso` y aceptar. El entorno de desarrollo se encarga de crear todo el esqueleto de la clase `Curso`.

```
//Curso.java
public class Curso {

    public void mostrarMensaje(){
        System.out.println("Bienvenidos al curso!!!");
    }
}

//Ejemplo19fp.java
public class Ejemplo19fp{

    public static void main(String[] args){
        Curso miCurso = new Curso(); //declara y crea objeto
        miCurso.mostrarMensaje(); //usa el objeto
    }
}
```

En memoria:

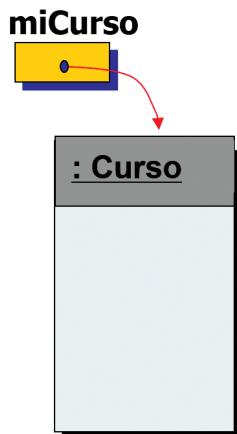


Figura 39. Estado de la memoria al crear el objeto `miCurso`

3.4. Atributos y métodos set y get

Los atributos son variables o constantes que se declaran fuera de todo método y se declaran antecediéndolos con alguno de los siguientes modificadores de acceso:

- `public`: para ser accesible por métodos dentro y fuera de la clase del objeto.
- `private`: para ser accesible sólo por métodos de la clase del objeto.
- `protected`: para ser accesible por métodos de la clase del objeto y descendientes.

Por recomendaciones de la Ingeniería de Software, los atributos deben ser generalmente **privados** (encapsulación) y los métodos deben ser **públicos**.

Recordar que los parámetros de los métodos son variables locales al método, pero los atributos existen mientras exista el objeto al que pertenecen y son accesibles desde cualquier método de su propia clase, pero para acceder a ellos desde fuera de su clase se hace por medio de los métodos públicos:

- `set` (mutador, para establecer/escribir el valor del atributo).
- `get` (accesor, para obtener/leer el valor del atributo).

EJEMPLO:

Crear una clase `Persona` con atributo `edad` y sus respectivos métodos `set` y `get`. Además crear una clase principal donde se cree un objeto de la clase `Persona` y donde se invoque a los dos métodos creados.

```
//Persona.java
public class Persona{

    private int edad;

    public void setEdad(int e){
        edad = e;
    }

    public int getEdad(){
        return edad;
    }
}

//Ejemplo20fp.java
public class Ejemplo20fp{

    public static void main(String[] args){

        Persona perl = new Persona();

        perl.setEdad(18);
        int suEdad = perl.getEdad();
        System.out.println(perl.getEdad());
    }
}
```

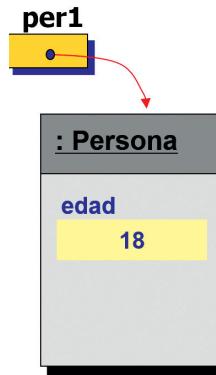


Figura 40. Estado de la memoria al crear el objeto `per1`

EJEMPLO:

Basado en el ejemplo del `Curso`, modificar las clases para que se cree el objeto `miCurso`, luego mostrar el valor del atributo `nombre` usando su accesror, después establecer un valor ingresado para dicho atributo usando su mutador, a continuación que imprima en pantalla el valor del atributo usando el método correspondiente `mostrarMensaje`.

```
//Curso.java
public class Curso{

    private String nombre;

    public void setNombre(String nom) {
        nombre = nom;
    }
    public String getNombre() {
        return nombre;
    }
    public void mostrarMensaje() {
        System.out.println("Bienvenidos al curso de "+nombre);
    }
}
//Ejemplo21fp.java
import java.util.*;

public class Ejemplo21fp{

    public static void main(String[] args) {
        String nombreCurso;
        Curso miCurso = new Curso();
        Scanner scan = new Scanner(System.in);

        System.out.println("Nombre curso:"+miCurso.getNombre());
        System.out.println("Ingresa el nombre del curso: ");
        nombreCurso = scan.nextLine();
        miCurso.setNombre(nombreCurso);
        miCurso.mostrarMensaje();
    }
}
```

3.5. Inicializando objetos con constructores

Un constructor es un método invocado automáticamente al momento de crear un objeto con el operador new y sirve generalmente para inicializar el objeto antes de usarlo.

Tiene el **mismo nombre** que la clase, no devuelve valores y es generalmente público.

Si no se incluye uno, Java crea un constructor por default (sin parámetros), pero si le creamos algún constructor explícitamente, Java ya no crea el constructor por default.

SINTAXIS:

```
public <nombreClase> (<parámetros> ) {
    <sentencias>
}
```

Es posible tener más de un constructor en una clase. Estos constructores son llamados constructores sobrecargados (overloaded) y siguen las mismas pautas que los métodos sobrecargados simples.

EJEMPLO:

Crear la clase `Curso` con constructor, set, get y `mostrarMensaje`. También crear la clase principal donde se creen dos objetos de `Curso` llamados `miCurso` y `otroCurso` que envíen sus respectivos nombres de curso como argumento al ser creados.

```
//Curso.java
public class Curso {

    private String nombre;

    public Curso(String nom) {
        nombre= nom; //también puede ir setNombreCurso(curso);
    }
    public void setNombre(String nom) {
        nombre = nom;
    }
    public String getNombre() {
        return nombre;
    }
    public void mostrarMensaje() {
        System.out.println("Bienvenidos al curso de "+nombre);
        //también puede ir
        //System.out.println("Bienvenidos al curso de "+getNombre());
    }
}
//Ejemplo22fp.java
public class Ejemplo22fp {

    public static void main(String[] args) {
        Curso miCurso = new Curso("Fundamentos de Programacion");
        Curso otroCurso = new Curso("Introduccion a Computacion");

        miCurso.mostrarMensaje();
        otroCurso.mostrarMensaje();
    }
}
```

3.6. Plantilla para definición de una clase

Recordar que sólo la clase principal tiene al método main. Las otras clases no.

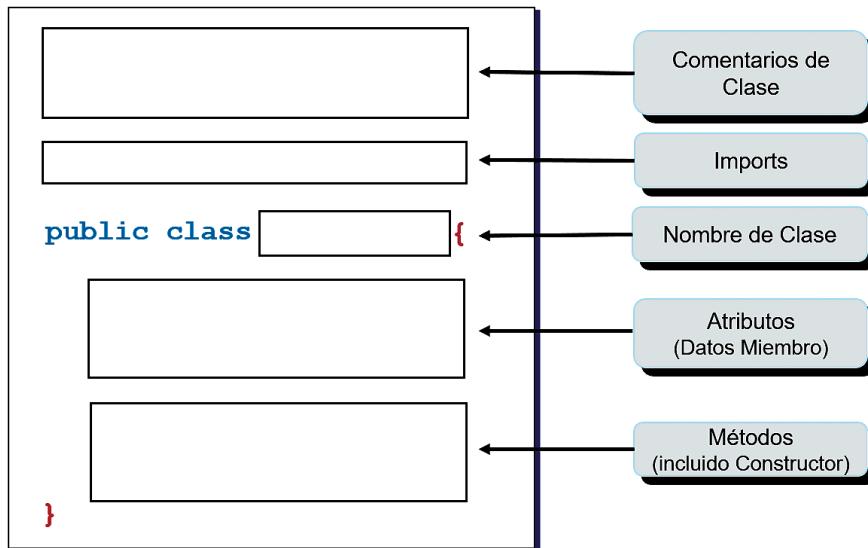


Figura 41. Plantilla de cualquier clase en Java

3.7. Métodos booleanos

Son métodos que revisan la verdad o falsedad de alguna condición del objeto, siempre retornan un valor boolean y normalmente empiezan con la palabra "is" o "es".

EJEMPLO:

Crear el método `esMenor` que determina si la edad de un objeto `Persona` es menor que 18. Suponer que la clase `Persona` tiene un atributo `edad`.

```

public boolean esMenor() {
    if (edad < 18)
        return true;
    else
        return false;
}
    
```

3.8. Método `toString`

Es un método que tienen todas las clases en Java y que retorna la información completa de un objeto (valores de sus atributos) en un `String`.

Es invocado automáticamente cuando utilizamos el nombre del objeto solo (sin métodos).

Dicho método puede ser sobrescrito para cualquier clase que creemos.

EJEMPLO:

```

//Curso.java
public class Curso{

    private String nombre;

    public Curso(String nom) {
        setNombre(nom);
    }
    public void setNombre(String nom) {
        nombre = nom;
    }
    public String getNombre() {
        return nombre;
    }
    public void mostrarMensaje() {
        System.out.println("Bienvenidos al curso de "+getNombre());
    }
    public String toString(){
        return "El nombre del curso es: "+nombre;
    }
}
//Ejemplo23fp.java
import java.util.*;

public class Ejemplo23fp{

    public static void main(String[] args){

        Curso miCurso = new Curso("Fundamentos de Programacion");
        Curso otroCurso = new Curso("Introduccion a Computacion");

        miCurso.mostrarMensaje();
        otroCurso.mostrarMensaje();

        System.out.println(miCurso); //invoca a toString
        System.out.println(miCurso+" y "+otroCurso); //a toString
    }
}

```

3.9. Diagrama de clases de UML

Ahora, nuestros programas ya no están constituidos por una sola clase (clase principal), sino además por otras clases.

Los diagramas de clases de UML nos permiten ver gráficamente la estructura de los programas creados.

Recordar que los rectángulos representan a las clases. Además, pueden haber flechas que muestran que hay algún tipo de relación entre dichas clases.

El símbolo + indica acceso público y el – indica acceso privado de los atributos y métodos.

EJEMPLO:

Realizar el diagrama de clases del ejemplo de los cursos.

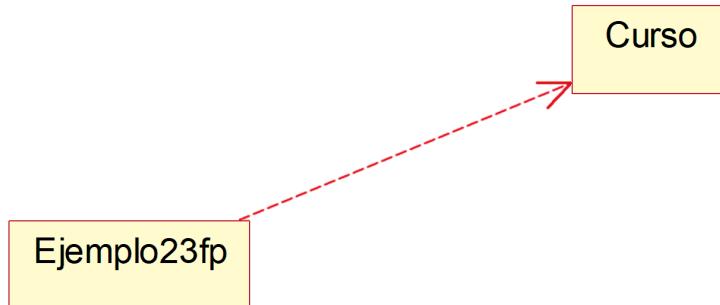


Figura 42. Diagrama de Clases de UML del ejemplo de los cursos

EJEMPLO:

Realizar el diagrama de clases completo del ejemplo de los cursos.

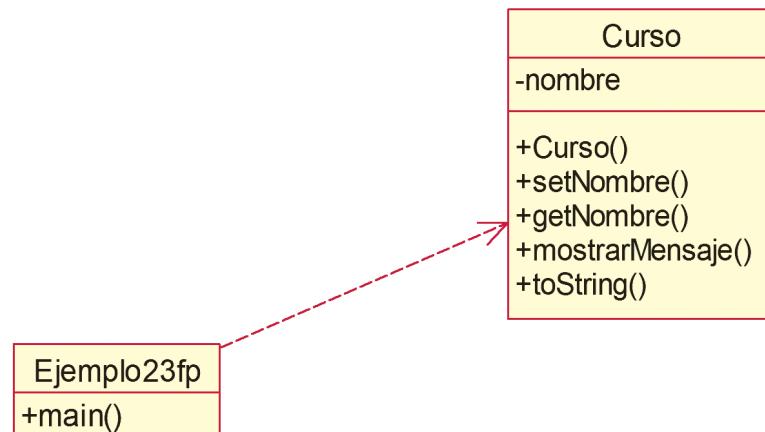


Figura 43. Diagrama de Clases de UML completo del ejemplo de los cursos

EJERCICIO 8:

Realizar el diagrama de clases de UML y el programa en que se simulan cuentas de ahorros donde podemos depositar y retirar.

Requerimientos:

- Sólo se trabaja con el tipo de moneda soles.
- También deberá brindar la funcionalidad de consultar el saldo de la cuenta.
- Probar el programa con 2 cuentas: depositar, retirar y que vaya mostrando sus datos al realizar cualquier movimiento en la cuenta.
- TIP: verificar que no se pueda retirar una cantidad si el saldo no es suficiente, mostrar mensaje de advertencia en dicho caso.

EJERCICIO 9:

Realizar un programa donde se simule el crecimiento de ratones recién nacidos para experimentos en un laboratorio de investigación en Biología. Cada ratón crece día a día y aumenta de peso, también día a día, usando la fórmula:

$$p_i = p_{i-1} + 0,1 * \text{tasa de Crecimiento} * \text{edad}$$

- p_i : peso del ratón el día i .
- El peso es en gramos y siempre empieza en 1 gramo, y la edad está en días (al nacer la edad es 0).
- La tasa de crecimiento es un porcentaje.
- También deberá brindar la funcionalidad de mostrar los datos del ratón.
- Probar el programa con 2 ratones, hacerlos crecer (solo se puede hacer crecer 1 día cada vez) y que vayan mostrando sus datos cada vez que el ratón crece.

3.10. Retornando un objeto de un método

Así como podemos retornar un valor primitivo de un método (`int`, `double`, `boolean`, etc.), también podemos retornar un objeto.

Cuando retornamos un objeto de un método, realmente estamos retornando una referencia al objeto (dirección de memoria).

Esto significa que no retornamos una copia del objeto, solamente la referencia a dicho objeto.

Se debe crear un objeto en dicho método y retornarlo. Para ello, debemos adecuar la cabecera del método para que retorne el tipo correspondiente del objeto.

EJEMPLO:

Crear la clase `Fraccion` que represente una fracción cualquiera. Con métodos `set` y `get` para sus atributos, `toString` y `simplifica` que devuelva una nueva fracción debidamente simplificada. Crear una clase principal que la valide.

```
public class Fraccion {
    private int numerador;
    private int denominador;

    public Fraccion(int num, int denom) {
        setNumerador(num);
        setDenominador(denom);
    }

    public void setNumerador(int num) {
        numerador = num;
    }

    public void setDenominador(int denom) {
        if (denom == 0) {
            System.err.println("Fatal Error");
            System.exit(1);
        }
        denominador = denom;
    }

    public int getNumerador() {
        return numerador;
    }
}
```

```

public int getDenominador( ) {
    return denominador;
}
public String toString( ){
    return getNumerador() + "/" + getDenominador();
}
public Fraccion simplifica( ){
    Fraccion simp;

    int num    = getNumerador();
    int denom = getDenominador();
    int mcd    = mcd(num, denom);

    simp = new Fraccion(num/mcd, denom/mcd);

    return simp;
}

public int mcd(int num, int denom){
    int menor = Math.min(num, denom);
    int valor = 0;
    int i = 1;

    while (i <= menor){
        if (num%i == 0 && denom%i == 0)
            valor = i;
        i++;
    }
    return valor;
}
}
public class Ejemplo24fp {

    public static void main(String[] args) {
        Fraccion f1,f2;

        f1 = new Fraccion(12,24);
        f2 = f1.simplifica();
        System.out.println(f1);
        System.out.println(f2);
    }
}

```

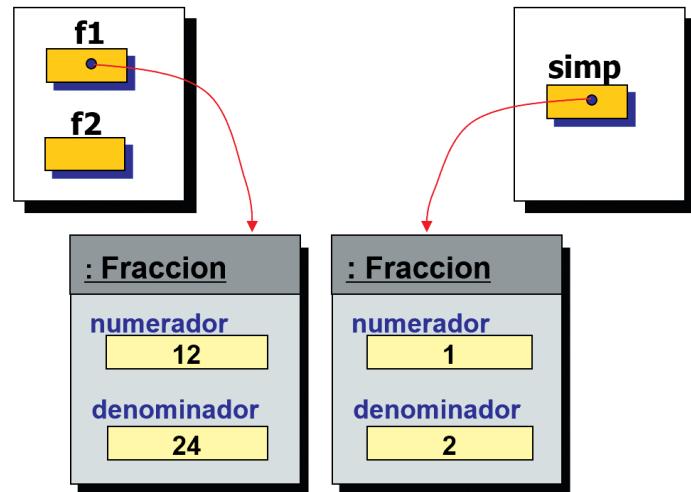


Figura 44. Estado de la memoria al entrar al método simplifica

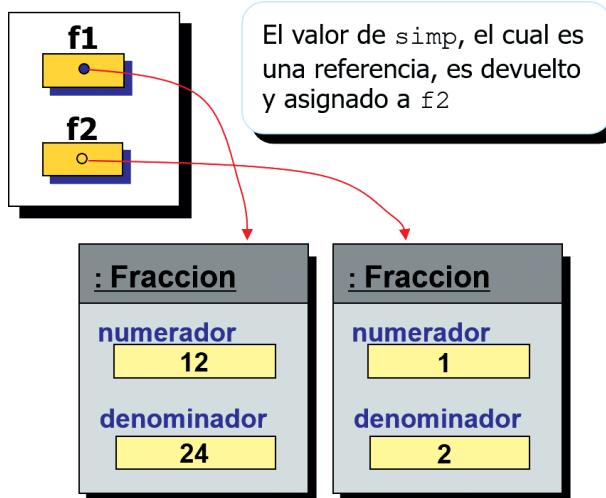


Figura 45. Estado de la memoria al retornar al método main

3.11. La referencia this

La palabra reservada `this` es llamada puntero de auto-referencia porque se refiere al mismo objeto llamado.

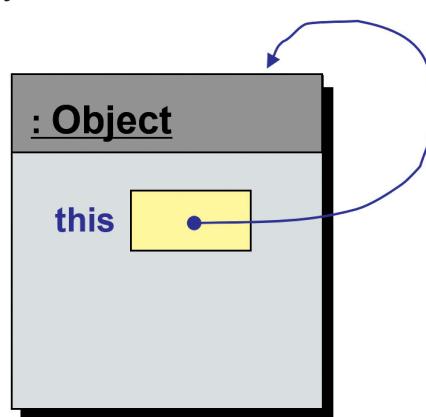


Figura 46. Referencia `this`

Se usa dentro de los métodos de una clase para referirse al objeto llamado (objeto sobre el que se invocó el método) o para hacer hincapié en él.

Es usado especialmente cuando se necesita distinguir entre un atributo y un parámetro que se llaman igual. En tal caso `this` se referirá al atributo.

También es útil porque es un nombre genérico, independiente del nombre específico del objeto llamado. Así, `this` sabe a qué objeto se refiere en un momento dado.

`this` puede ser usado cuando sea necesario o también por legibilidad.

EJEMPLO:

Observar la clase `Persona` que tiene un atributo `edad` y dentro del método `setEdad` tiene un parámetro que se llama también `edad`. Para distinguirlos se debe usar `this` para referirse al atributo o dato miembro.

```
public class Persona {  
    int edad;  
  
    public void setEdad(int edad) {  
        this.edad = edad; //this.edad se refiere al atributo  
                         //edad se refiere al parámetro  
    }  
    . . .  
}
```

EJEMPLO:

Observar la clase `Persona` que tiene los atributos `nombre` y `edad`. Aquí no sería obligatorio usar `this` para referirse al atributo o dato miembro, pero se puede hacer por legibilidad.

```
public class Persona {  
    private String nombre;  
    private int edad;  
  
    public void setDatos(String nom, int e) {  
        this.nombre = nom; //usar this por legibilidad  
        this.edad = e; //podría ser edad = e;  
    }  
    . . .  
}
```

EJERCICIO 10:

Crear la clase `Coordenada` que representa un punto en el plano cartesiano (x, y), con su constructor y métodos necesarios.

Que tenga 3 formas de calcular la distancia entre 2 coordenadas (distancia euclíadiana):

- Método en clase principal
- Método en clase `Coordenada` con 2 parámetros
- Método en clase `Coordenada` con 1 parámetro

3.12. Constructores sobrecargados

Recordar que dos o más métodos pueden tener el mismo nombre siempre y cuando cumplan al menos alguna de las siguientes reglas:

- 1) Tengan un numero diferente de parámetros.
- 2) Si el número de parámetros es el mismo, sus parámetros deben ser de tipos de dato diferentes.

EJEMPLO:

Regla 1:

```
public void miMetodo(int x, int y) { ... }
public void miMetodo (int x) { ... }
```

Regla 2:

```
public void suma(double x, double y) { ... }
public void suma(int x, int y) { ... }
```

Las mismas reglas se aplican para los constructores sobrecargados, cuando tenemos dos o más constructores (mismo nombre y parámetros diferentes).

EJEMPLO:

Regla 1:

```
public Tiempo( ) { ... }
public Tiempo(int hora, int minuto, int segundo) { ... }
```

Regla 2:

```
public Persona(int dni) { ... }
public Persona(String name) { ... }
```

Para llamar un constructor sobrecargado desde otro constructor usar:

```
this(<argumentos para el constructor Destino>);
```

Esta llamada al constructor puede aparecer sólo en la definición de los métodos constructores, y solamente como la **primera sentencia** de dicha definición.

EJEMPLO:

Crear la clase Fraccion que corresponda a la siguiente clase principal. Usar la llamada de constructor sobrecargado a otro constructor sobrecargado.

```
public class Ejemplo25fp {

    public static void main(String[] args) {
        Fraccion a = new Fraccion(3, 4);
        Fraccion b = new Fraccion(3);
        Fraccion c = new Fraccion();
        Fraccion d = new Fraccion(a);
        System.out.println(a);
        System.out.println(b);
```

```
        System.out.println(c);
        System.out.println(d);
    }
}
```

La clase Fraccion correspondiente sería la siguiente, observar el uso de this:

```
public class Fraccion {
    private int numerador;
    private int denominador;

    public Fraccion( ) {
        this(1, 1); //crea 1/1
    }

    public Fraccion(int num) {
        this(num, 1); //crea num/1
    }

    public Fraccion(Fraccion frac) {
        //crea copia de Fraccion
        this(frac.getNumerador(), frac.getDenominador());
    }

    public Fraccion(int num, int denom) {
        setNumerador(num);
        setDenominador(denom);
    }

    public void setNumerador(int num) {
        numerador = num;
    }

    public void setDenominador(int denom) {
        if (denom == 0) {
            System.err.println("Fatal Error");
            System.exit(1);
        }
        denominador = denom;
    }

    public int getNumerador( ) {
        return numerador;
    }

    public int getDenominador( ) {
        return denominador;
    }

    public String toString( ){
        return getNumerador() + "/" + getDenominador();
    }
}
```

3.13. Valores por default de los atributos

Es el valor que tiene un atributo cuando no es inicializado explícitamente.

Esta situación sólo sucede con los atributos, porque cuando creamos variables comunes no existen valores por default (lo que almacenan cuando no son inicializados explícitamente es un valor “basura”).

EJEMPLO:

Declaración de los atributos de la clase `Persona`, aquí están inicializados explícitamente:

```
private int edad = 0;
private double peso = 2.0;
private double estatura; //su valor por default es 0.0
```

Tipo de dato	Valores por default
Entero	0
Real	0.0
Boolean	false
Tipo por referencia (objeto)	null

Tabla 4. Valores por default de atributos dado su tipo de dato

3.14. Ambigüedad de nombres de variables

Un identificador que aparece dentro de un método puede ser una variable local, un parámetro o un atributo (dato miembro).

En general, cuando el compilador encuentra un identificador, primero busca coincidencia dentro del método y luego dentro de la clase.

Las reglas para un identificador encontrado en un método son:

- 1) Si hay una declaración de variable local o un parámetro que coincide, el identificador se refiere a él.
- 2) De otra manera, si no hay variables locales coincidentes, pero si hay una declaración de atributo que coincide, el identificador se refiere a dicho atributo.
- 3) De otra forma, es un error porque no hay una declaración que coincida.

3.15. Atributos constantes

Es posible que algunos atributos de una clase sean constantes, siendo necesarios cuando requerimos atributos cuyos valores no deben variar.

Se siguen las mismas recomendaciones de estilo que se usan para constantes comunes. Utilizaremos la palabra reservada `final` y nombres en mayúsculas con guiones bajos como separadores de palabras.

Hay que tener en cuenta que al crear un atributo constante, se necesita crear un objeto para poder utilizarla, al igual que se hace con atributos no constantes.

SINTAXIS:

```
<modificadorAcceso> final <tipoDato> <nombre> = <valorInicial>;
```

EJEMPLO:

```
private final String NOMBRE_PROPIETARIO = "Anonimo";
```

Definir constantes brinda una descripción significativa de los valores, evitando "números mágicos" (literales).

EJEMPLO:

Se define el siguiente atributo constante en una clase:

```
private final int INDEFINIDO = -1;
```

Luego en un método cualquiera, escribir:

```
numero = INDEFINIDO;
```

es más significativo que escribir:

```
numero = -1;
```

Brinda facilidad de mantenimiento. Sólo necesitamos cambiar el valor en la declaración de la constante en lugar de buscar todas las ocurrencias del mismo valor en el código.

3.16. Encadenamiento de llamadas a métodos (Method-Call Chaining)

Hasta ahora hemos llamado a un método del objeto a la vez:

```
miAuto.setMarca("Honda");  
miAuto.setAnio(2003);
```

Pero también tenemos la posibilidad de encadenar 2 ó más llamadas a métodos:

```
miAuto.setMarca("Honda").setAnio(2003);
```

Es el mecanismo denominado "encadenamiento de llamadas a método". Usamos un punto para concatenar una llamada a un método al final de una llamada a otro método. Se evalúa de izquierda a derecha, pero debemos preparar los métodos para que retornen un objeto y para eso utilizamos la referencia `this` para retornar el objeto llamado.

EJEMPLO:

Crear la clase Auto y prepararla para usar encadenamiento de métodos.

```
public class Auto {  
  
    private String marca;  
    private int anio;
```

```
public Auto setMarca(String m) {
    marca = m;
    return this;
}
public Auto setAnio(int y) {
    anio = y;
    return this;
}
public void imprimir() {
    System.out.println(marca + ", " + anio);
}
}

public class Ejemplo26fp {

    public static void main(String[] args) {
        Auto miAuto = new Auto();
        miAuto.setMarca("Honda").setAnio(1998).imprimir();
    }
}
```

En los métodos `setMarca` y `setAnio` de `Auto`, notar como se permite el encadenamiento de llamadas a métodos.

Así, para lograr dicho funcionamiento, en cada definición de dichos métodos:

- 1) Se retorna el objeto llamado (que servirá para la siguiente llamada):

```
return this;
```

- 2) Cada cabecera del método debe especificar la clase del objeto que se retornará (que servirá para la siguiente llamada):

```
public Auto setMarca(String m)
```

Tenemos la posibilidad de aplicar este mecanismo para crear código más compacto y fácil de entender.

3.17. Atributos y métodos de clase

En la mayoría de los casos hemos utilizado atributos y métodos de instancia, que significa que debemos crear objetos (instancias) de la clase para poder invocar sus métodos y manipular sus atributos.

Pero hay ocasiones en las que es más conveniente usar atributos y métodos de clase, en los que **no es necesario** crear objetos (instancias) para invocar sus métodos y manipular sus atributos.

En este caso se invocan sus métodos y atributos directamente de la clase correspondiente.

EJEMPLO:

Todos los métodos y atributos que hemos utilizado hasta el momento de la clase `Math` son métodos y atributos de clase.

La clase `Math` contiene el atributo de clase `PI` y el método de clase `round`. `PI` y `round` están asociados con toda la clase `Math`, no con una instancia en particular de dicha clase. Se invocan directamente con el nombre de clase, sin necesidad de crear objetos.

```
System.out.println(Math.PI);
System.out.println(Math.round(3.1415));
```

Para crear miembros de clase usamos el modificador `static`.

SINTAXIS:

```
<modificadorAcceso> static <tipoDato> <nombre>;           //atributo
<modificadorAcceso> static <tipoDato> <nombre>(<listaparámetros>)//método
```

EJEMPLO:

```
private static int valMin = 1;
private static int valMax = 6;
public static int sumar(int a, int b) { ... }
```

3.18. Atributos constantes de clase

Son atributos de clase que además son constantes. Son compartidos por todos los objetos de la misma clase y son accedidos directamente por medio de la clase. Para crearlos utilizaremos los modificadores `static` y `final` y como modificador de acceso es muy común utilizar `public` ya que al ser constantes no necesitan ser protegidas de cambios no deseados.

`PI` y `E` de la clase `Math` son atributos constantes de clase.

SINTAXIS:

```
<modificadorAcceso> static final <tipoDato> <nombre> = <valorInicial>;
```

EJEMPLO:

```
private static final String NOMBRE_PROPIETARIO = "Anonimo";
```

3.19. Clases de utilidad

Son clases que implementan métodos que son de propósito más general (son aplicables en diferentes proyectos).

A dichos métodos se les llama métodos de utilidad. Por ejemplo, tenemos los métodos de la clase `Math` previamente vistos, tales como:

```
Math.round()           Math.sqrt()
```

Las clases de utilidad contienen típicamente sólo constantes y métodos de clase.

Los miembros de una clase de utilidad son generalmente `public` y `static` para que otras clases puedan acceder fácilmente a ellos.

Incluso las constantes son públicas, porque al ser constantes su valor no se verá afectado nunca y ya no sería necesario encapsularlas como privadas. Por ejemplo, las constantes de la clase `Math` son accedidas desde cualquier programa en Java.

```
Math.PI
```

```
Math.E
```

EJEMPLO:

Crear la clase de utilidad UtilidadesImpresion que permita imprimir una cadena centrada y subrayada. También crear una clase principal que la use.

```

public class UtilidadesImpresion {

    public static final int MAX_COL = 80;
    public static final int MAX_FIL = 50;

    public static void imprimirCentrado(String s) {
        int colInicial;
        colInicial = (MAX_COL / 2) - (s.length() / 2);

        for (int i = 0; i < colInicial; i++)
            System.out.print(" ");

        System.out.println(s);
    }

    public static void imprimirSubrayado(String s) {
        System.out.println(s);
        for (int i = 0; i < s.length(); i++)
            System.out.print("-");

        System.out.println();
    }
}

public class Ejemplo27fp {

    public static void main(String[] args) {
        System.out.println(UtilidadesImpresion.MAX_COL);
        System.out.println(UtilidadesImpresion.MAX_FIL);
        UtilidadesImpresion.imprimirCentrado("Hola Amigos");
        UtilidadesImpresion.imprimirSubrayado("Que tal");
    }
}

```

3.20. Los miembros de una Clase

En resumen, podemos agrupar a todos los miembros que puede tener una clase como atributos o métodos, y éstos a su vez se pueden agrupar en miembros de clase o de instancia según como se creen.

Constantes de clase
Constantes de instancia
Variables de clase
Variables de instancia

Constructores()
Métodos de clase()
Métodos de instancia()

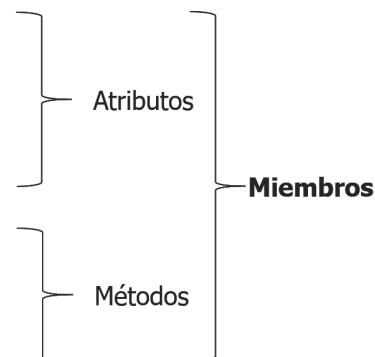


Figura 47. Clasificación de todos los miembros de una Clase

EJERCICIOS PROPUESTOS

Para cada problema crear las clases indicadas y además crear un main que utilice objetos de dichas clases.

1. Crear la clase Rectángulo que representa a la figura geométrica, nos importa calcular su área, su perímetro y verificar si es cuadrado. Además, crear los métodos set y get para cada atributo y 3 métodos constructores sobrecargados.
2. Crear la clase Cuenta que representa una cuenta bancaria, nos importa el nombre del titular de la cuenta y su saldo. Puedo depositar, retirar, ver saldo. Además, crear los métodos set y get para cada atributo, crear constructor y esPremium (que verificará si la cuenta tiene saldo mayor o igual a 10000).
3. Crear la clase Hora que representa una hora del día, por ejemplo 3:15:25. Crear los siguientes métodos: un constructor para inicializar la hora con valores consistentes, crear los métodos setUnaHora para establecer una hora completa (hh, mm, ss) y que llame a setMinuto, setSegundo y setHora para que verifiquen los datos correctos. Además, los métodos aumentarMinuto, aumentarSegundo, aumentarHora. También crear los métodos get para cada atributo y mostrarHora que imprima la hora en formato hh:mm:ss.
4. Crear la clase Fecha con 6 constructores sobrecargados: 1/1/1900, 1/1/a, 1/m/a, d/m/a, copia de f, x días transcurridos desde el 1/1/2000. Además crear los métodos set, que verifiquen la consistencia de sus datos y den valores consistentes en caso sean erróneos, también crear los métodos get y toString. Ver que serán necesarios otros métodos como esBisiesto.
5. Crear la clase Fraccion con constructores sobrecargados, los cuales verifiquen la consistencia de sus datos y le den valores por defecto en caso sean erróneos. Crear métodos set y get. También que permita mostrar en diferentes formatos: quebrado (a/b) y decimal (x.yzw). Y que permita retornar la suma, resta, multiplicación y división de la fracción con otro objeto fracción (el objeto llamado queda intacto). Además, que permita retornar la fracción simplificada (el objeto llamado queda intacto). Evitar la duplicación de código.

```
suma = f1.sumar(f2);
```

6. Crear la clase Fraccion como en el anterior ejemplo, pero que permita el encadenamiento de llamadas a métodos (f1 puede cambiar su valor).

```
f = f1.sumar(f2).sumar(f3).restar(f4).mult(f5);
```

CAPÍTULO 4

AGREGACIÓN, COMPOSICIÓN Y HERENCIA

OBJETIVOS:

- Comprender como las cosas son organizadas naturalmente mediante los mecanismos de agregación y composición
- Implementar las relaciones de agregación y composición en un programa
- Comprender como usar la herencia para reutilizar una clase ya creada
- Implementar la herencia en un programa
- Aprender a construir constructores para las clases derivadas y cómo sobrescribir un método heredado
- Solucionar problemas aplicando los conceptos avanzados de agregación, composición y herencia

4.1. Introducción

Hasta el momento sólo hemos visto la relación de Dependencia entre clases. Dicha relación indica que la `ClasePrincipal` hace uso o depende de la `OtraClase` para poder funcionar.

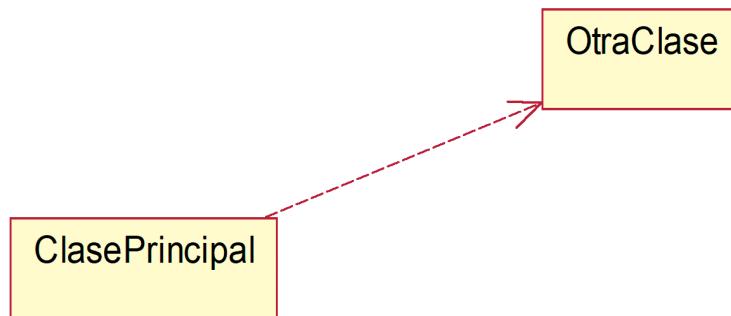


Figura 48. Relación de dependencia entre clases

A continuación, veremos otros tipos de relaciones entre clases.

4.2. Agregación y composición

Anteriormente, todos nuestros objetos fueron relativamente simples, así pudimos declarar cada objeto del tipo de una sola clase individual.

Para objetos más complejos, deberíamos considerar dividir el objeto en sus partes y definir una clase como el **todo** y otras clases como las **partes** del todo.

Cuando la clase **todo** está compuesta de las clases **parte**, la organización de clases se llama **agregación**.

Cuando la clase **todo** es además la **propietaria exclusiva** de las clases **parte**, la organización de clases se llama **composición**.

Ambas se llaman relación "TIENE" o "TIENE - UN".

La relación de agregación es más débil que la composición ya que no hay exclusividad entre las clases que participan en la relación.

Con la agregación una clase es el todo y otras clases son partes del todo (como con la composición), pero **no hay la restricción adicional** que requiere que las partes sean exclusivas del todo.

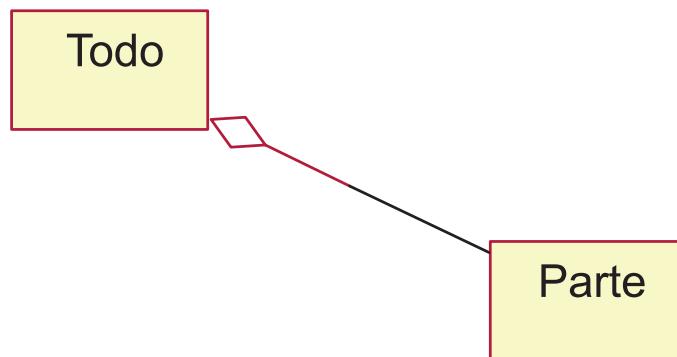


Figura 49. Relación de Agregación de UML

EJEMPLO:

- Podemos implementar una Escuela Profesional como una agregación, creando la clase todo para la Escuela y las clases parte para los diferentes tipos de personas que trabajan y estudian en la Escuela.
- Las personas no son propiedad exclusiva de la Escuela, porque una persona puede ser parte de más de una agregación.
- Así, una persona puede pertenecer a 2 escuelas diferentes y ser parte de 2 agregaciones.

El concepto de agregación y composición no es nuevo, es lo que hacemos con los objetos complejos en el mundo real.

EJEMPLO:

- Cada ser viviente y casi todo producto está hecho de partes.
- Muchas veces, cada parte a su vez está hecha de su propio conjunto de subpartes.

Recordar que una relación de **composición**, es exactamente igual que una relación de agregación, pero en la composición la parte está limitada a tener un sólo propietario a la vez.

EJEMPLO:

- Un corazón puede estar en sólo un cuerpo a la vez.
- Un motor puede estar en sólo un vehículo a la vez.

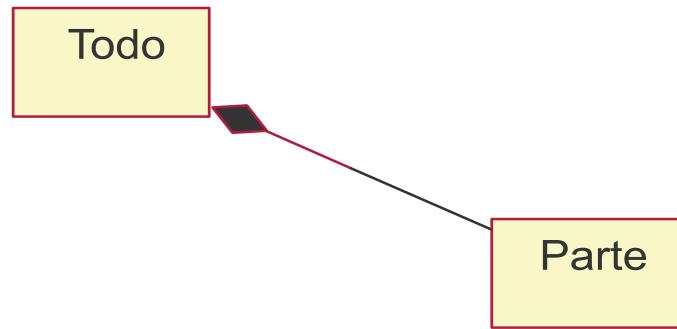


Figura 50. Relación de Composición de UML

EJEMPLO:

Tratamos de modelar un Concesionario de Autos.

El Concesionario **tiene** un grupo de autos, un gerente de ventas y un grupo de vendedores.

¿Qué clases necesitamos?

Auto, Gerente, Vendedor, Concesionario.

El Concesionario contiene a las otras 3 clases.

Existen relaciones del Concesionario con las otras 3 clases, ¿qué tipo de relación tiene?

Tomemos en cuenta las siguientes suposiciones:

- Auto sólo pertenece a un Concesionario.
- Gerente puede dirigir varios Concesionarios.
- Vendedor puede trabajar para varios Concesionarios.

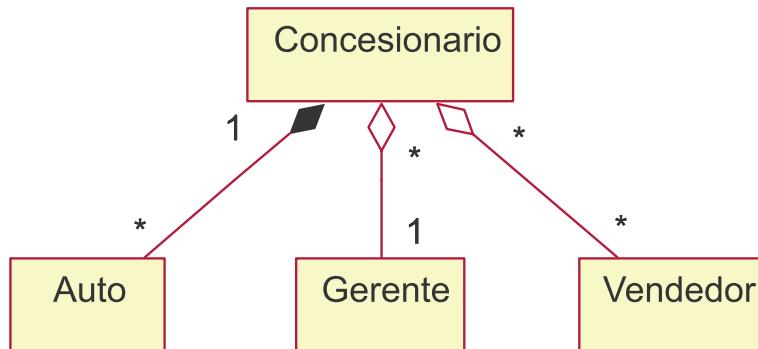


Figura 51. Diagrama de Clases UML del ejemplo planteado

Entre dos clases pueden existir las siguientes relaciones:

- Línea sólida entre 2 clases es una asociación genérica (relación más simple). Pero que puede evolucionar a cualquiera de las siguientes relaciones.
- Línea discontinua es relación de dependencia o uso.
- Línea sólida con rombo lleno es una composición.
- Línea sólida con rombo vacío es una agregación.
- El rombo va hacia el lado del todo.
- En el diagrama anterior, verificar que los números y signos en las líneas son

los valores de multiplicidad (indica el número de instancias que participan en la relación entre 2 clases).

- * representa a "muchos", también se puede usar N.
- Para establecer la multiplicidad de cada relación se hace la prueba bidireccional. Se inicia considerando 1 en un lado de la relación y se pregunta ¿cuántos participarán en el otro extremo?, la respuesta se coloca como multiplicidad en el otro extremo. Posteriormente se hace la lectura en la otra dirección para establecer la multiplicidad en el punto de inicio.
 - Un Concesionario puede administrar **muchos** Autos (se pone *)
 - Un Auto puede ser administrado por **un** Concesionario (se pone 1)
 - Relación uno a muchos se representa: 1 – muchos, 1 – *, 1 – N

Para implementar un programa usando agregación y composición:

- Definir una clase para el **todo** y otras clases para cada **parte**.
- Para una clase **contenedora**, declarar un atributo tal que almacene la referencia de uno o más de los objetos **contenidos**.
- Típicamente la multiplicidad * (en la clase **parte**) usa un ArrayList u otra estructura de datos que permita almacenar varios objetos para ser implementada.
- Si dos clases tienen la relación de **agregación**, almacenar el objeto de la clase contenida por medio de una variable de referencia en la clase contenedora. Pero además, almacenarla en otra variable de referencia fuera de la clase contenedora, así dicho objeto puede ser añadido a otra agregación y puede tener 2 ó más propietarios (no hay exclusividad).
- Si dos clases tienen una relación de **composición**, almacenar el objeto de la clase contenida en una variable de referencia en la clase contenedora, pero **no almacenarla en otro lugar**. Así tendrá 1 sólo propietario.

EJEMPLO:

Crear el diagrama de clases UML y el programa en Java. Crear un concesionario con su gerente, 2 vendedores y 3 autos. Además, se desea imprimir el estatus del concesionario con el siguiente formato de salida:

Nombre de la empresa, del gerente, los datos de los vendedores y de los autos

El programa debe permitir añadir vendedores y autos al concesionario.

Cada auto sólo pertenece a un concesionario, pero un gerente o vendedor puede pertenecer a varios.

Por lo pronto y para simplificar el problema, usar como atributos para cada clase:

- Auto: marca
- Gerente: nombre
- Vendedor: nombre, ventas

Modelamos en UML las clases correspondientes con sus atributos, métodos, relaciones y multiplicidades. Ver figura anterior. Recordar que el Concesionario **tiene** un grupo de autos, un gerente de ventas y un grupo de vendedores.

```

public class Auto {
    private String marca;

    public Auto(String m) {
        marca = m;
    }

    public String getMarca() {
        return marca;
    }
}

public class Gerente {
    private String nombre;

    public Gerente(String n) {
        nombre = n;
    }

    public String getNombre() {
        return nombre;
    }
}

public class Vendedor {
    private String nombre;
    private double ventas = 0.0; //ventas realizadas

    public Vendedor(String n) {
        nombre = n;
    }
    public String getNombre() {
        return nombre;
    }
}

import java.util.*;

public class Concesionario {
    private String nombreEmpresa;
    private Gerente elGerente;
    private ArrayList<Vendedor> personal = new ArrayList<Vendedor>();
    private ArrayList<Auto> autos = new ArrayList<Auto>();

    public Concesionario(String nombre, Gerente geren) {
        nombreEmpresa = nombre;
        elGerente = geren;
    }

    public void addAuto(Auto a) {
        autos.add(a);
    }
}

```

```

public void addVendedor(Vendedor ven) {
    personal.add(ven);
}

public void imprimirEstatus() {
    System.out.println("Nombre Empresa    Nombre Gerente");
    System.out.println(nombreEmpresa + " " + elGerente.getNombre());
    System.out.println("Vendedores:");
    for (Vendedor vendedor : personal) {
        System.out.println(vendedor.getNombre());
    }
    System.out.println("Autos:");
    for (Auto auto : autos) {
        System.out.println(auto.getMarca());
    }
}
}

public class Ejemplo28fp {

public static void main(String[] args) {
    Gerente elGerente = new Gerente("Juan Gomez");
    Vendedor jenny = new Vendedor("Jennifer Tapia");
    Vendedor alan = new Vendedor("Alan Perez");
    Concesionario miEmpresa = new Concesionario("EPIS", elGerente);

    miEmpresa.addVendedor(jenny);
    miEmpresa.addVendedor(alan);
    miEmpresa.addAuto(new Auto("BMW a"));
    miEmpresa.addAuto(new Auto("Tico x"));
    miEmpresa.addAuto(new Auto("Combi y"));
    miEmpresa.imprimirEstatus();
}
}

```

4.3. Herencia

Hasta el momento hemos visto las relaciones de dependencia, agregación y composición.

La herencia es otro tipo de relación, también llamada relación "ES - UN" o de especialización.

Los seres humanos usamos este mecanismo en diferentes áreas de la ciencia ya que nos permite manejar la complejidad.

EJEMPLO:

Clasificación de los seres vivos en la naturaleza.

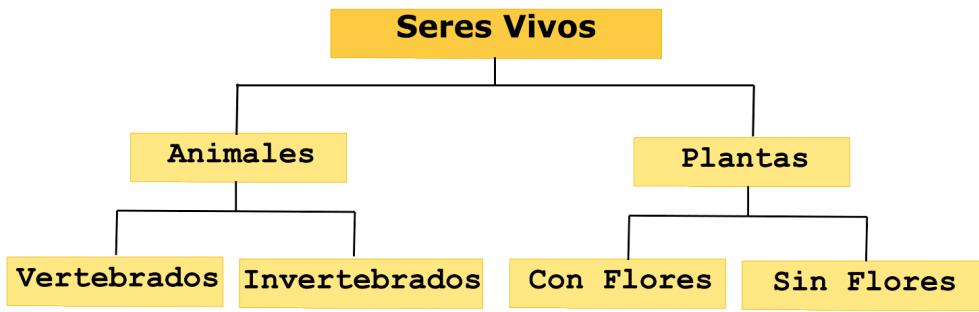


Figura 52. Una posible clasificación de los seres vivos

Esta relación se representa en UML:

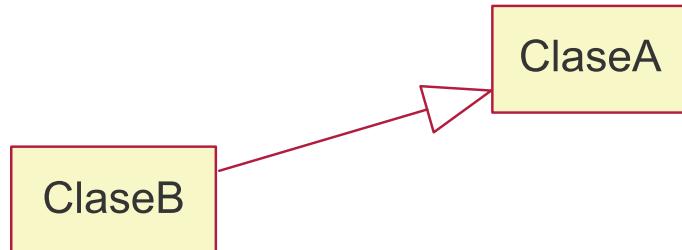


Figura 53. Relación de Herencia de clases en UML

- ClaseA: superclase, parente, base o generalizada.
- ClaseB: subclase, hija, derivada o especializada.
- Ancestro: parente, parente del parente...
- Descendiente: hijo, hijo del hijo...

EJEMPLO:

Imaginen que queremos hacer un software para una clínica veterinaria que se encarga de tratar gatos y perros.

Ambos son diferentes... pero ¿tienen similitudes?

Si, ¡ambos son mascotas!, así podemos decir que perro **es una** mascota y que gato **es una** mascota. Algunos datos comunes pueden ser nombre, edad y color.

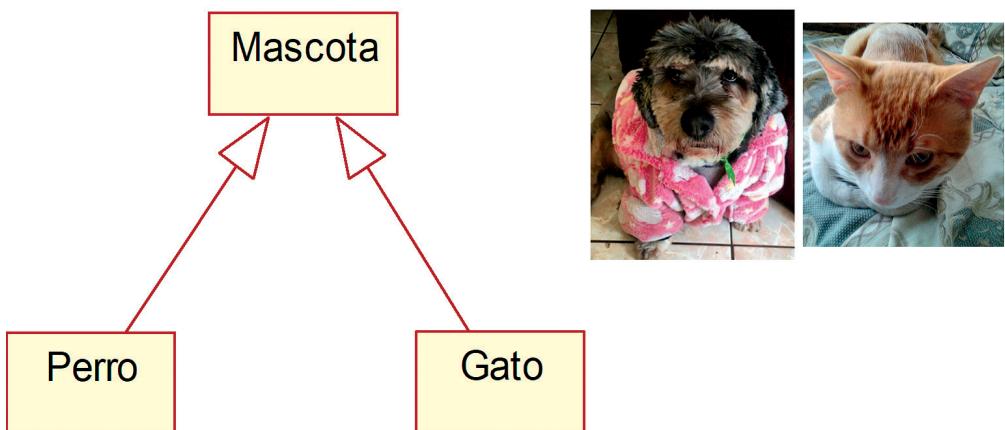


Figura 54. Diagrama de clases en UML considerando la herencia

EJEMPLO:

Suponga que Ud. está a cargo del diseño de autos en una empresa automotriz que tiene 5 modelos de autos y se le pide que haga los planos para cada modelo, así podría tomar 2 enfoques para realizar la tarea encomendada:

- 1) Podría crear planos de diseño independientes para cada modelo. Repetiría la misma tarea varias veces, ya que todos los modelos se parecen en gran parte.
- 2) Pero para ahorrar tiempo es mejor crear un plano maestro que describa las características comunes de todos los modelos.

Después hacer planos adicionales, uno por cada modelo, que describan las características específicas de cada modelo.

Utilizar el segundo enfoque es análogo al concepto de herencia, donde una nueva clase es derivada de una clase existente.

Se llama herencia porque la nueva clase hereda todas las características y comportamientos (atributos y métodos) de la clase existente.

Ésta es una característica muy importante de Java y de la POO, porque permite a los programadores reutilizar el código existente y evitar duplicar código (ahorando recursos y dando calidad al programa creado).

- El código de una clase puede ser reutilizado por múltiples subclases.
- Esto elimina la redundancia de código, permitiendo crear código más mantenible para corregir y evolucionar.
- Un programador puede reutilizar una clase existente para crear fácilmente una nueva subclase (sin reinventar nada).

La clase existente sirve como base de todas las clases derivadas de ella.

Crear una subclase de algo que funciona perfectamente ayuda a conservar lo bueno y probar lo nuevo.

Además, permite trabajar con módulos más pequeños (ya que las clases se dividen en superclases y subclases), haciendo así, más fácil el mantenimiento del código.

EJERCICIO 11:

Establecer la jerarquía de clases y realizar el diagrama de clases UML de:

- Rectángulo, cuadrilátero, paralelogramo, trapecio.

EJERCICIO 12:

Para cada una de las siguientes clases indicar posibles clases derivadas:

- Estudiante.
- FiguraGeométrica.
- CuentaBancaria.
- MiembroComunidadUniversitaria.

SINTAXIS:

```
public class <nombreClase> extends <nombreSuperClase> {  
    ...  
}
```

Indica que la clase hereda de la superclase.

EJEMPLO:

¿Cómo sería la cabecera de la clase Círculo que hereda de la clase FiguraGeometrica?

```
public class Círculo extends FiguraGeometrica {
    ...
}
```

EJEMPLO:

Dado el siguiente diagrama de clases UML, crear el programa en Java. Además, crear la clase principal donde se cree objetos de las clases mostradas en el diagrama UML y los utilice.

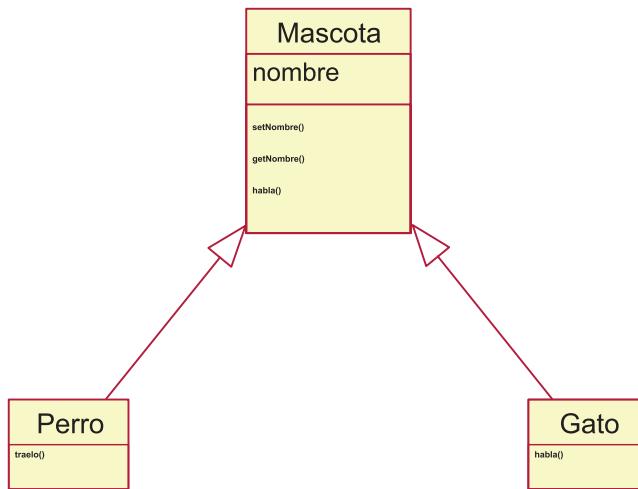


Figura 55. Diagrama de clases detallado en UML considerando la herencia

```
public class Ejemplo29fp {

    public static void main(String[] args) {
        Perro myDog = new Perro();
        Gato myCat = new Gato();

        System.out.println(myDog.habla());
        System.out.println(myDog.traelo());
        System.out.println(myCat.habla());
        //System.out.println(myCat.traelo()); //error
    }
}

public class Mascota {
    private String nombre;

    public void setNombre(String s) {
        nombre = s;
    }

    public String getNombre() {
        return nombre;
    }
}
```

```
public String habla() {  
    return "Yo soy tu sumisa mascota!";  
}  
}  
public class Gato extends Mascota{  
  
    public String habla() {  
        return "No me des ordenes, hablare cuando lo deseé!!!!";  
    }  
}  
public class Perro extends Mascota{  
  
    public String traeLo() {  
        return "Si maestro, traerlo debo!";  
    }  
}
```

4.4. Llamada al constructor de una superclase

A diferencia de otros métodos, los constructores de la superclase no son heredados por la subclase.

Un constructor puede llamar al constructor de su superclase usando `super`.

```
super (<argumentos>);
```

Una llamada a un constructor está permitida sólo si es la **primera línea** en el cuerpo del constructor.

4.5. Sobrescritura de métodos (Overriding)

Una subclase puede sobrescribir un método de su superclase, así tendrá un método con la misma cabecera que el método de su superclase, aunque el cuerpo cambie.

Por default, un objeto de la subclase usará su propio método (no el métodos de su superclase). Cuando un objeto de la subclase necesite llamar al método sobrescrito de la superclase deberá usar `super.nombreMetodo()`.

EJERCICIO 13:

Realizar la implementación en Java de las clases del diagrama de clases de UML y de una clase principal que cree y use un objeto de la clase `TiempoCompleto`.

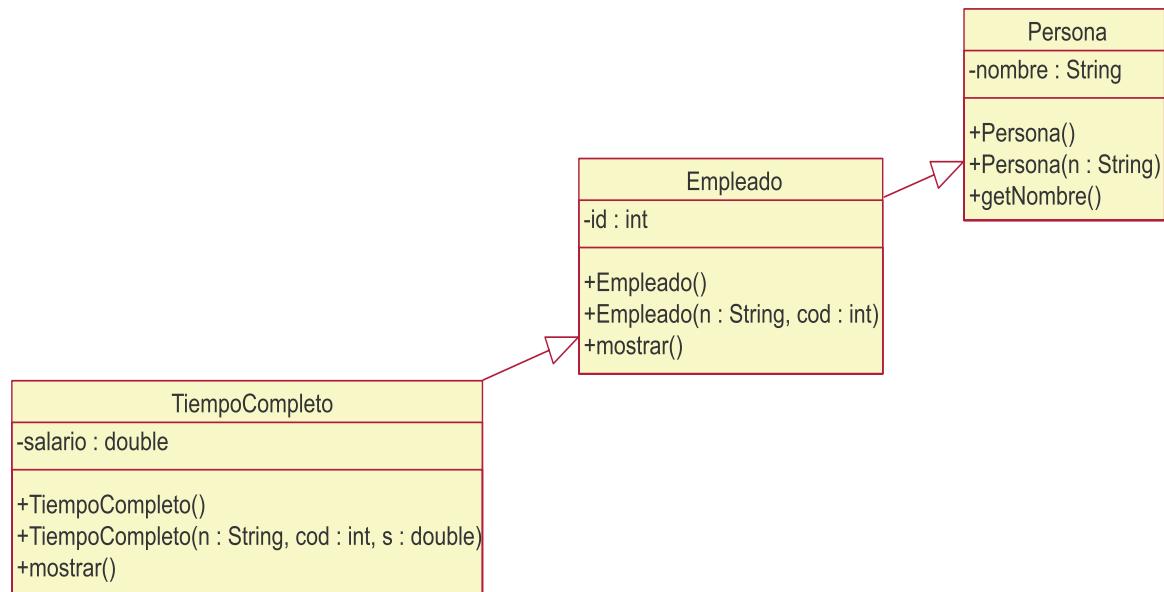


Figura 56. Diagrama de clases en UML considerando la herencia

4.6. Comparando la agregación, composición y herencia

Agregación y composición: cuando una clase es el todo y otras clases son partes de ese todo. Relación "TIENE UN". Son similares pero la composición es exclusiva.

Herencia: cuando una clase es una versión más detallada de otra clase. La subclase hereda los miembros de la superclase (atributos y métodos). Relación "ES UN".

Es muy común tener combinadas todas estas relaciones en una aplicación.

EJERCICIO 14:

Para el programa `Concesionario` aplicar Ingeniería Inversa y crear su diagrama de clases de UML. ¿Qué relación de herencia podríamos añadir al programa?.

Rehacer el diagrama de clases considerando la nueva relación de herencia y rehacer el programa.

EJERCICIOS PROPUESTOS

1. Crear un diagrama de clases de UML para un programa simulador de juego de cartas (sólo la representación de la baraja, todavía no el juego) y el código en Java de dichas clases, luego validar con alguna clase principal.

Seguir los siguientes pasos:

- Decidir sobre las clases apropiadas para resolver el problema.
- Para cada clase dibujar su representación UML.
- Buscar relaciones de agregación/composición entre las clases.
- Dibujar las relaciones donde convenga.
- Para cada clase decidir sobre sus atributos apropiados y ponerlos en el diagrama.
- Para cada clase decidir sobre sus métodos apropiados y ponerlos en el diagrama.
- Buscar atributos y métodos comunes.
- Si 2 ó más clases contienen un conjunto de atributos o métodos comunes, implementar una superclase y mover los miembros comunes a la superclase.
- Crear una relación de herencia de dichas clases con la superclase.
- Modificar el diagrama de clases.
- Crear las clases en Java.

2. Crear un diagrama de clases de UML para un programa de juego de cartas y el código en Java de dichas clases, luego validar con alguna clase principal.

Basarse en el ejercicio anterior y seguir los siguientes pasos:

- El juego es de tipo “guerra” donde tenemos una baraja de cartas y 2 jugadores. Se reparten las cartas, mitad a cada uno y luego cada jugador muestra una carta, el que tenga el número mayor se lleva la carta del otro. Gana quien deja sin cartas al otro.
- Decidir sobre las clases apropiadas para resolver el problema.
- Para cada clase dibujar su representación UML.
- Buscar relaciones de agregación/composición entre las clases.
- Dibujar las relaciones donde convenga.
- Para cada clase decidir sobre sus atributos y métodos apropiados y ponerlos en el diagrama.
- Buscar atributos y métodos comunes.
- Si 2 ó más clases contienen un conjunto de atributos o métodos comunes, implementar una superclase y mover los miembros comunes a la superclase.
- Crear una relación de herencia de dichas clases con la superclase.
- Modificar el diagrama de clases y crear las clases en Java.

CAPÍTULO 5

HERENCIA, POLIMORFISMO Y TÓPICOS AVANZADOS

OBJETIVOS:

- Comprender el rol de la clase `Object` como superclase de todas las clases en Java
- Sobrescribir los métodos `toString` y `equals` en clases propias
- Comprender como el polimorfismo y el enlace dinámico mejoran la versatilidad del programa
- Entender las restricciones de asignar un objeto de una clase a la variable de referencia de otra clase
- Comprender cómo usar un arreglo de variables de referencia de un ancestro para implementar el polimorfismo
- Verificar cómo una declaración `abstract` en la superclase elimina la necesidad de definir métodos tontos
- Comprender cómo se crean y usan las clases `interface`
- Entender el modificador de acceso `protected`
- Solucionar problemas aplicando los conceptos avanzados de la orientación a objetos

5.1. La clase `Object`

La clase `Object` es la superclase para todas las clases en Java.

Cuando declaramos nuestras propias clases, no tenemos que especificar que heredarán de `Object`, lo hace automáticamente.

Veremos 2 métodos de `Object` que ya hemos usado: `equals` y `toString`.

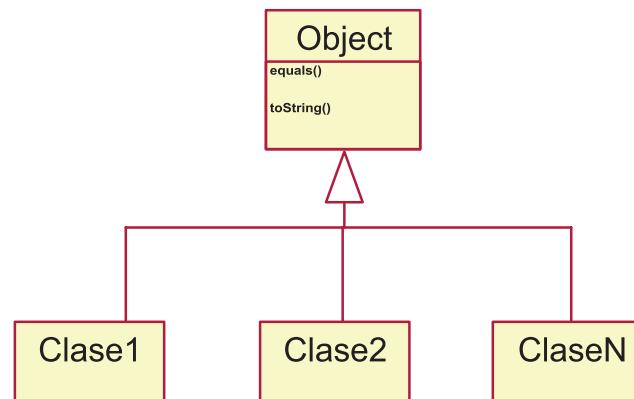


Figura 57. Diagrama de clases considerando una jerarquía de la clase `Object`

5.2. El método equals

Dada una clase que no tiene su propio método `equals`, si tenemos objetos de dicha clase pueden ser comparados con el método `equals`.

Esto es posible porque ellos ya heredan y usan el método `equals` de la clase `Object`.

Dicho método retorna `true` si las 2 variables de referencia que están siendo comparadas apuntan al mismo objeto (contienen la misma referencia o dirección).

El `equals` heredado trabaja exactamente igual que el operador `==`, compara referencias, no contenidos.

Generalmente el método `equals` de la clase `Object` no es tan útil.

Casi siempre necesitamos comparar el **contenido** de 2 objetos en lugar de verificar que dos variables de referencia apuntan al mismo objeto.

Para lograrlo deberíamos **sobrescribir** el método `equals` en nuestras clases, de tal forma que compare el contenido de los objetos.

EJEMPLO:

Escribir el método `equals` de la clase `Auto` creada previamente para que compare contenidos de 2 objetos `Auto` y comprobar su funcionamiento.

Luego poner dicho método `equals` como comentario y verificar el funcionamiento del programa.

```
public class Ejemplo30fp {
    public static void main(String[] args) {
        Auto a1 = new Auto("Toyota", 1992, "Verde");
        Auto a2 = new Auto("Toyota", 1992, "Verde");
        if(a1.equals(a2))
            System.out.println("Son iguales");
        else
            System.out.println("Son diferentes");
    }
}
public class Auto {
    private String marca;
    private int anio;
    private String color;

    public Auto (String m, int a, String c){
        setMarca(m);
        setAnio(a);
        setColor(c);
    }
    public void setMarca(String m) {
        marca = m;
    }
    public void setAnio(int a){
        anio = a;
    }
    public void setColor(String c) {
        color = c;
    }
}
```

```
public boolean equals(Auto otro) {
    return otro != null && marca.equals(otro.marca) &&
           anio == otro.anio && color.equals(otro.color);
}
```

El método `equals` está implementado en muchas clases de la API de Java.

Por ejemplo, en la clase `String` y en las clases `Wrapper`.

En la clase `String`, `equals` comprueba si los contenidos de los 2 objetos `String` que son comparados son los mismos.

5.3. El método `toString`

Retorna una cadena que describe al objeto.

El método `toString` de la clase `Object` retorna una cadena que es una concatenación del nombre completo de la clase del objeto (incluye nombre del paquete de la clase), el signo @ y una secuencia hexadecimal llamada `hashcode`.

EJEMPLO:

Considerar el código:

```
Object obj = new Object();
System.out.println(obj.toString());

Auto a1 = new Auto();
System.out.println(a1.toString());
```

Salida:

```
java.lang.Object@601bb1
Auto@1ba34f2
```

Lo anterior no es tan útil, por lo que llamar al método `toString` de la clase `Object` tampoco lo es.

Debemos sobrescribir el método `toString` en nuestras propias clases.

En general el método `toString` debería retornar un `String` que describe el contenido del objeto llamado (valores de sus atributos concatenados).

Se encuentran muchos métodos `toString` sobrescritos en las clases de la API de Java.

Por ejemplo, la clase `Date` tiene un método `toString` que retorna los valores concatenados en un `String` del mes, día y año.

Ya que recuperar los contenidos de un objeto es una tarea muy común, deberíamos tener el hábito de proveer un método `toString` para la mayoría o todas las clases propias.

Típicamente el método `toString` debería simplemente concatenar los valores de los atributos del objeto y devolverlos como un `String`.

Notar que el método `toString` no debería imprimir el `String` concatenado.

Recordar que si se imprime o se concatena usando sólo el nombre del objeto, se está llamando implícitamente a su método `toString`.

EJEMPLO:

Aumentar el método `toString` a la clase `Auto` creada previamente y comprobar su funcionamiento.

```
public String toString() {  
    return "marca=" + marca + ", anio=" + anio + ", color=" + color;  
}
```

En la clase principal aumentar al final:

```
System.out.println(a1); //invocará al método toString creado
```

5.4. Métodos `toString` de las clases Wrapper

Todas las clases wrapper tienen métodos `toString` que retornan un valor `String` del valor primitivo dado.

EJEMPLO:

```
String s1 = Integer.toString(22); //convierte a String "22"  
String s2 = Double.toString(123.45); //convierte a String "123.45"  
System.out.println(s1 + " " + s2);
```

También tenemos la opción de usar el método `valueOf` de la clase `String`.

EJEMPLO:

```
String s3 = String.valueOf(22);  
String s4 = String.valueOf(123.45);  
String s5 = String.valueOf('S');  
char[] vocales = {'a', 'e', 'i', 'o', 'u'};  
String s6 = String.valueOf(vocales);
```

5.5. Polimorfismo

Es cuando diferentes tipos de objetos responden de forma diferente a la misma llamada a método.

Para implementarlo: declarar una variable de referencia de un tipo general que sea capaz de referirse a objetos de diferente tipo (objetos de subclases del tipo general). Para hacerlo usamos una **superclase**.

¿Cuál superclase usaríamos sin necesidad de crear una clase propia? La superclase `Object`.

El polimorfismo permite que una variable de referencia del tipo de una superclase se refiera a objetos de diferentes subclases de dicha superclase.

EJEMPLO:

Dado el ejemplo de la clase `Mascota` (superclase), `Gato` y `Perro` (subclases), el siguiente código sería válido:

```
Mascota miMascota = new Mascota();  
miMascota = new Perro();  
miMascota = new Gato();
```

EJEMPLO:

Imagine ahora que en el ejemplo de las mascotas no existe la clase `Mascota`.

Se declarará `obj` como un `Object` y la llamada a método `obj.toString()` es la que exhibe un comportamiento polimórfico (llamada polimórfica).

- Si `obj` referencia a un objeto `Perro`, `toString` retorna “Guau! Guau!”
- Si `obj` referencia a un objeto `Gato`, `toString` retorna “Miau! Miau!”

Las líneas resaltadas muestran la llamada al método polimórfico `toString`.

Dependiendo del objeto que se cree, `Perro` o `Gato`, se llamará al `toString` de su clase correspondiente.

Ver que no es necesario que la variable de referencia y el objeto sean del mismo tipo. Es válido que el objeto sea de un tipo que sea subclase del tipo de la variable de referencia.

```
import java.util.*;  
  
public class Ejemplo31fp {  
  
    public static void main(String[] args) {  
        Scanner scan = new Scanner(System.in);  
        Object obj;  
        System.out.print("Que tipo de mascota deseas? "  
                        + "Ingresá p para perros y g para gatos: ");  
        if (scan.next().equals("p"))  
            obj = new Perro();  
        else  
            obj = new Gato();  
  
        System.out.println(obj.toString());  
        System.out.println(obj);  
    }  
  
    public class Gato {  
        public String toString() {  
            return "Miau! Miau!";  
        }  
    }  
  
    public class Perro {  
        public String toString() {  
            return "Guau! Guau!";  
        }  
    }  
}
```

EJEMPLO:

Dada la jerarquía de clases verificar el siguiente código:

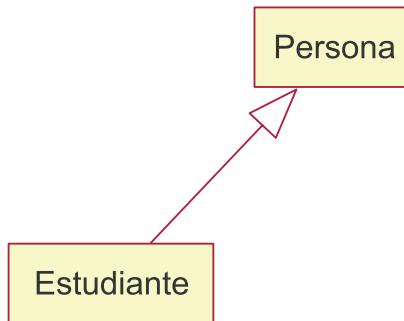


Figura 58. Diagrama de clases considerando una jerarquía de clases de herencia

```

Persona p = new Estudiante(); //correcto, Estudiante es una persona
Estudiante s = new Persona(); //incorrecto, Persona no es Estudiante
  
```

5.6. Enlace Dinámico (Dynamic Binding)

El polimorfismo es un concepto, una forma de comportamiento. Pero el enlace dinámico es el mecanismo que permite realizar dicho comportamiento.

Es lo que la Máquina Virtual de Java (JVM) hace para corresponder una llamada a un método polimórfico con un método en particular.

Justo antes de ejecutar la llamada al método, la JVM determina el tipo de objeto que ha sido asignado a la variable de referencia.

Si es de la clase x, la JVM enlaza el método correspondiente a x a la llamada al método original.

Después que la JVM enlaza el método apropiado a la llamada al método, la JVM ejecuta el método enlazado.

Tener en cuenta que cuando un compilador ve la llamada a método <variableReferencia>.<método>(), revisa si la clase de la variable de referencia tiene la definición del método llamado, si no la encuentra, no compila.

Normalmente, al asignar un objeto a una variable de referencia, la clase de ambos es la misma, pero en el ejemplo anterior vimos que un objeto de la clase Perro es asignado en una variable de referencia de tipo Object. Tal asignación sólo funciona si la clase del lado derecho (objeto) es una subclase de la clase del lado izquierdo (variable de referencia). En este caso Gato y Perro son subclases de Object.

EJEMPLO:

```

Object obj = new Object();
Object obj = new Perro();
Perro obj = new Object(); //no válido
  
```

5.7. Operador instanceof

Es un operador que verifica si un objeto referenciado es instancia de una clase en particular.

Devuelve true si el objeto corresponde con su clase, con su superclase o con cualquiera de sus ancestros.

SINTAXIS:

```
<nombreObjeto> instanceof <nombreClase>
```

EJEMPLO:

```
Object obj = new Perro();
System.out.println(obj instanceof Perro); //true
System.out.println(obj instanceof Gato); //false
System.out.println(obj instanceof Object); //true

if(obj instanceof Perro)
    System.out.println("Es un perrito");
```

5.8. Polimorfismo con arreglos

La utilidad del polimorfismo viene cuando tenemos un arreglo de objetos (o cualquier estructura de datos que pueda contener objetos). Dicho arreglo almacena variables de referencia de objetos y le asignamos diferentes tipos de objetos a los elementos del arreglo o estructura de datos.

El polimorfismo nos permite recorrer el arreglo y llamar al método polimórfico para cada elemento del arreglo.

En tiempo de ejecución, la JVM usa el enlazamiento dinámico para elegir el método en particular a llamar para los diferentes tipos de objetos del arreglo.

EJEMPLO:

Verificar como un arreglo de tipo **Mascota** permite referenciar objetos de **Mascota** y de sus subclases.

```
Mascota[] misMascotas = new Mascota[40];

misMascotas[0] = new Mascota();
misMascotas[1] = new Gato();
misMascotas[2] = new Perro();
```

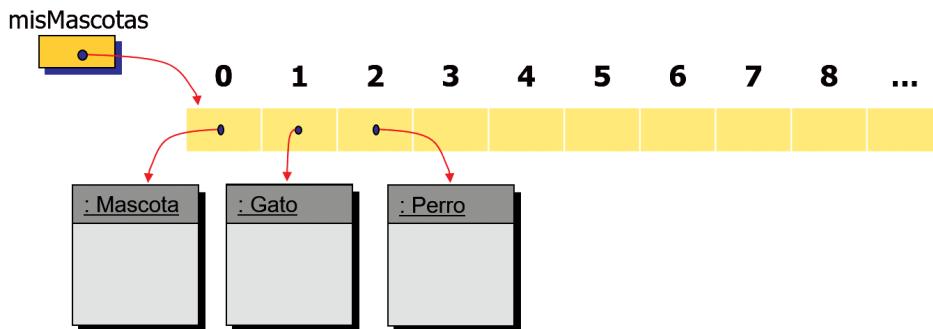


Figura 59. Arreglo de Mascotas que permite objetos de subclases

EJEMPLO:

Verificar en siguiente código cómo se puede utilizar el arreglo creado:

```

for (int i = 0; i < misMascotas.length; i++)
    System.out.println(misMascotas[i].getNombre());

for (int i = 0; i < misMascotas.length; i++)
    misMascotas[i].aumentarEdad();

int contadorGatos = 0;
for (int i = 0; i < misMascotas.length; i++)
    if ( misMascotas[i] instanceof Gato )
        contadorGatos++;

```

EJEMPLO:

En la empresa tenemos 2 tipos de Empleado: el empleado que tiene un salario fijo mensual y el empleado que trabaja por horas (el salario fijo y el pago por hora varía de empleado a empleado).

Deseamos manejar la planilla y obtener el reporte de cuánto pagar a fin de mes para cada empleado.

Añada 160 horas, pero sólo a cada objeto de empleado que gana por horas (antes de la impresión de sus datos).

Crear las clases necesarias. Implementar el Diagrama UML y la clase principal que pruebe las clases creadas. Aplicar polimorfismo.

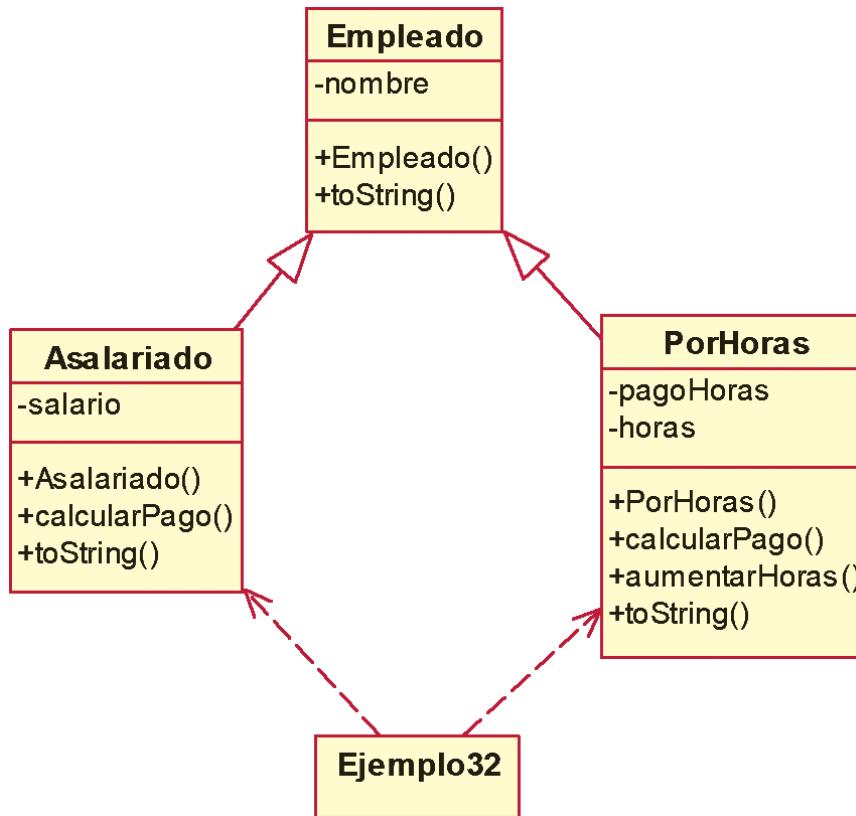


Figura 60. Diagrama de clases para el problema

```

public class Ejemplo32fp {

    public static void main(String[] args) {

        Empleado[] empleados = new Empleado[100];
        PorHoras empleadoPorHoras;
        empleados[0] = new PorHoras("Carla", 25.0);
        empleados[1] = new Asalariado("Juan", 48000);
        empleados[2] = new PorHoras("Pedro", 20.0);

        for (int i=0;i< empleados.length && empleados[i] != null;
i++) {
            if (empleados[i] instanceof PorHoras) {
                empleadoPorHoras = (PorHoras) empleados[i];
                empleadoPorHoras.aumentarHoras(160);
            }
            System.out.println(empleados[i]); //llamada polimórfica
        }
    }

    public class Empleado {
        private String nombre;

        public Empleado(String n) {
            nombre = n;
        }

        public String toString() {
            return nombre;
        }
    }

    public class PorHoras extends Empleado{
        private double pagoHora;
        private double horas = 0.0;

        public PorHoras(String n, double pago) {
            super(n);
            pagoHora = pago;
        }

        public double calcularPago() {
            double salario = pagoHora * horas;
            return salario;
        }

        public void aumentarHoras(double h) {
            horas += h;
        }

        public String toString() {
            return super.toString() + " " + calcularPago();
        }
    }

    public class Asalariado extends Empleado{
        private double salario;
    }
}

```

```

public Asalariado(String n, double s) {
    super(n);
    salario = s;
}
public double calcularPago() {
    return salario;
}
public String toString() {
    return super.toString() + " " + calcularPago();
}
}

```

5.9. Métodos y clases abstract

Definir una clase como `abstract` implica indicarle al compilador que no se permite crear instancias de dicha clase.

Si un programa intenta instanciar una clase `abstract`, se genera un error de compilación.

Básicamente se utiliza para las superclases en una jerarquía de herencia de clases, donde las subclases son las que en realidad se instancian.

Persona

Figura 61. Representación UML de una clase abstracta (cursiva)

Declarar un método como `abstract` si la clase del método es una superclase y el método es un “método dummy” que será sobrescrito en las subclases.

Una clase abstracta puede tener métodos abstractos y concretos.

Un método `abstract` no debería ser `private` ni `final` ya que se va a acceder desde fuera de la clase y va a tener que ser sobrescrito.

Java requiere que cuando definamos un método como `abstract`, debemos:

- Usar una cabecera de método `abstract` en lugar de una definición completa del método.
 - Igual que una cabecera de método, excepto que incluye el modificador `abstract` y acaba con ;

```
public abstract double getPay();
```

- Definir una versión sobrescrita de dicho método en cada una de las subclases de la superclase.
- Definir la superclase como `abstracta` usando el modificador `abstract`.

EJEMPLO:

Verificar en el siguiente código cómo se crea una clase abstracta y un método abstracto.

```
public abstract class Empleado{

    private String nombre;
    public abstract double getPago();

    public Empleado(String n){
        nombre = n;
    }
    public void mostrarPago(){
        System.out.println(nombre +" "+ getPago());
    }
}
```

5.10. Interfaces

Una interface es como una clase abstracta pura.

Sólo provee 2 tipos de miembros:

- 1) Constantes de clase: aunque no es necesario poner los modificadores `public static final`. Permite acceder a ellas directamente desde las clases que implementen dicha interface.
- 2) Declaraciones de método: aunque no es necesario poner `public abstract`. Los métodos deben ser sobrescritos por todas las clases que implementen dicha interface.

SINTAXIS:

```
interface <nombreInterface>{
    <tipo> <nombreConstante> = <valor>;
    ...
    <tipoRetorno> <nombreMétodo>(<listaParámetros>);
    ...
}
```

EJEMPLO:

```
interface Comision{
    double TASA_COMISION = 0.10;
    void aumentarVentas(double ventas);
}
```



Figura 62. Representación UML de una interface (estereotipo `<<interface>>`)

Las interfaces deben ser implementadas por otras clases para que sean útiles.

Una clase puede implementar una interface:

```
public class <nombreClase> implements <nombreInterface>{  
    ...  
}
```

Una clase también puede implementar varias interfaces:

```
public class <nombreClase> implements <nomInter1>, <nomInter2>,  
    ...{  
    ...  
}
```

Una clase puede usar herencia junto a implementación:

```
public class <nombreClase> extends <nombSuper> implements <nomInter1>,  
< nomInter2>, ...{  
    ...  
}
```

Se usan las interfaces con diferentes fines:

- 1) Almacenar constantes universales.
- 2) Implementar polimorfismo.
- 3) Estandarizar la comunicación interclases (Ingeniería de Software).

EJEMPLO:

```
public class Comisionado implements Comision{  
  
    private String nombre;  
    private double ventas = 0.0;  
  
    public Comisionado(String nom) {  
        nombre = nom;  
    }  
  
    public void [aumentarVentas](double s) { //se declaró en Comision  
        ventas += s;  
    }  
  
    public double calcularPago(){  
        double pago = [TASA COMISIÓN] * ventas; //se declaró en Comision  
        return pago;  
    }  
}
```

5.11. Herencia vs interfaces

La herencia es usada para compartir código común (atributos y métodos) entre las instancias de clases relacionadas.

Las interfaces son usadas para compartir comportamiento común entre las instancias de diferentes clases no necesariamente relacionadas.

Si una clase es una especialización de otra, entonces especificarla como su

subclase aplicando la herencia.

Considerar que Java sólo maneja herencia simple (cada clase sólo puede tener una superclase), pero puede implementar n interfaces.

5.12. Modificador de acceso protected

El modificador de acceso `protected` está a un nivel intermedio entre los modificadores de acceso `public` y `private` en términos de cuán accesible es un miembro de la clase.

Los miembros `protected` (atributos y métodos de instancia y de clase) sólo pueden ser accedidos directamente dentro de su misma clase o por clases descendientes (subclases).

¿Cuándo usar modificador de acceso `protected`?

Cuando deseamos que un miembro sea accesible desde cualquier clase dentro de su subárbol de clases (descendientes) en una jerarquía de clases, pero no deseamos que sea accesible desde algún otro lugar.

EJEMPLO:

Suponga que la clase `Empleado` contiene un atributo `id` y todas sus clases descendientes necesitan acceder a él.

Aunque aún se podrá acceder por medio de los métodos mutador y accesos correspondientes, se puede hacer más fácil el acceso para sus descendientes al declarar `id` como `protected`:

```
protected int id;
```

Así, las clases descendientes pueden acceder al atributo `id` directamente (como si fuera un atributo propio) en lugar de usar su método accesos `getId()` o su método mutador `setId()`.

EJERCICIOS PROPUESTOS

Para cada problema crear las clases indicadas y además crear un `main` que utilice objetos de dichas clases.

1. Crear el diagrama de clases de UML y programa:

En un software graficador tenemos 3 tipos de figuras: Triángulo, Rectángulo y Círculo.

Crear una superclase para dichas figuras.

Aplicar polimorfismo creando una sola variable de referencia de la clase padre y usando los métodos polimórficos `toString` (muestra datos miembro y área) y `getArea` (que muestre su respectiva área).

2. En la empresa tenemos 4 tipos de Empleado: el que tiene un salario fijo mensual, el que trabaja por horas (el pago por hora varía de empleado a empleado), el que trabaja por comisión (10% de sus ventas) y el que tiene un salario fijo mensual, pero también comisión.

Deseamos manejar la planilla y obtener el reporte de cuánto pagar a fin de mes para cada empleado.

Implementar el Diagrama UML y una clase principal que pruebe las clases creadas. Usar polimorfismo e Interfaces.

- Crear un arreglo de 100 Empleados.
 - Crear 5 empleados como elementos del arreglo: 2 por horas, 1 con salario, 1 por comisión y 1 por salario y comisión.
 - Darle una cantidad de horas trabajadas a los empleados por hora.
 - Darle una cantidad de total de ventas a los empleados por comisión.
 - Imprimir el reporte de cuánto se les debe pagar.
3. Utilizar el nivel de acceso `protected` en miembros de la clase `Asalariado`. La misma empresa vista anteriormente, pero con la siguiente restricción:
 - Existen empleados a los que se les debe descontar cuando se les paga.
 - Los empleados afectados son aquéllos que tienen como sueldo fijo 10000 ó más, la tasa de descuento es del 15% (sobre el sueldo fijo).

CAPÍTULO 6

PROGRAMACIÓN ORIENTADA A EVENTOS E INTERFACE GRÁFICA DE USUARIO (GUI)

OBJETIVOS:

- Comprender el paradigma de Programación Orientado a Eventos (Event - Driven)
- Describir los principales paquetes GUI que brinda la API de Java
- Usar la clase JFrame para implementar la funcionalidad de una ventana
- Crear y usar los componentes: JLabel, JTextField, JButton, JTextArea, JCheckBox, JRadioButton y JComboBox
- Implementar la interface ActionListener para brindar comportamiento a los componentes
- Comprender lo que es una clase interior (inner class)
- Crear y usar cuadros de diálogo JOptionPane
- Usar la clase Color para darle una nueva dimensión a las GUI
- Solucionar problemas aplicando el paradigma Orientado a Eventos

6.1. Introducción

Se define GUI como Graphical User Interface (Interface gráfica de Usuario).

- Gráfica: objetos visuales (ventanas, botones, menús, etc.).
- Usuario: persona que usa el programa.
- Interface: medio por el que el usuario interactúa con el programa.

Aunque las empresas todavía escriben programas basados en texto (tipo ventana de comandos), la mayoría escribe programas basados en GUI para los productos que desarrollan para el mercado.

Los programas visuales (con GUI o con ventanas) usualmente usan técnicas de programación orientada a eventos.

La idea básica detrás de este paradigma de programación es:

- El programa espera que ocurran eventos y luego responde a ellos.
- Un evento es un mensaje que le dice al programa que algo ha ocurrido.
- Por ejemplo: el usuario presionó un botón, entonces se genera un evento y le dice al programa que un botón en particular fue presionado.
- Formalmente: cuando un usuario clica un botón, decimos que el botón dispara un evento.

EJEMPLO:

Acciones y reacciones de nuestra interacción diaria con el software:

Acción: presionar la tecla ENTER mientras el cursor está en un cuadro de texto.

- El cuadro de texto dispara un evento y le dice al programa que la tecla ENTER se presionó.

Acción: hacer click en una opción del menú principal.

- La opción del menú dispara un evento y le dice al programa que se seleccionó dicha opción.

Acción: cerrar una ventana presionando el botón X.

- La ventana dispara un evento indicando que se presionó el botón de cierre.

6.2. Fundamentos de la programación orientada a eventos

Si un evento se dispara y deseamos que nuestro programa lo maneje, entonces necesitamos crear un **listener** para el evento.

EJEMPLO:

Si deseamos que nuestro programa haga algo cuando un usuario cliquee un botón en particular, entonces necesitamos crear un **listener** para el botón.

Si no hay un **listener** para un evento disparado, entonces el evento no será escuchado y no habrá respuesta a él.

Si lo hay, el **listener** escuchará el evento y el programa responderá al evento disparado.

El programa responderá ejecutando código conocido como **manejador de evento** (**event handler**).

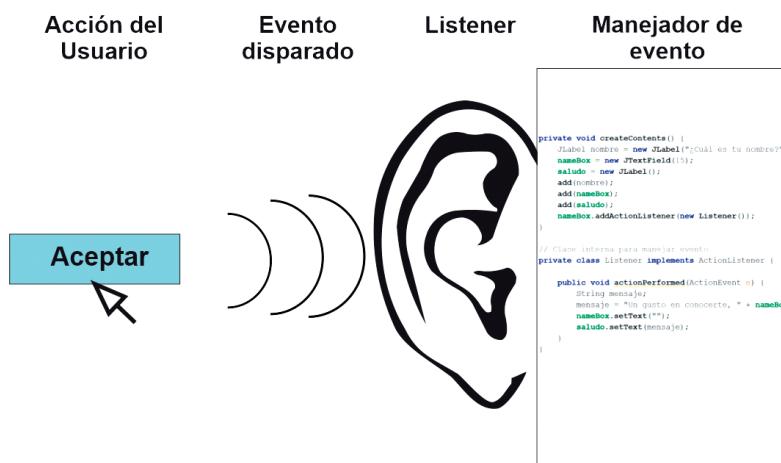


Figura 63. Programación Orientada a Eventos

6.3. Clases y paquetes GUI

Usaremos las diferentes clases de Java para GUI, tales como:

- Clase `JButton` cuando necesitemos un botón.

- Clase `Color` cuando necesitemos especificar un color.

En el JDK 1.0, todas las clases GUI estaban en Abstract Windowing Toolkit (AWT).

Eran poco ideales en términos de portabilidad, así Sun desarrolló componentes GUI más portables y los puso en la biblioteca GUI llamada Swing.

En la actualidad se usan ambas:

- El paquete para Swing es `javax.swing`
- El paquete para AWT es `java.awt` y `java.awt.event`

EJEMPLO:

Crear una ventana cuyo título sea “Ventana de ejemplo” de tamaño 300x200 y que contenga una etiqueta con el texto “Esta es una etiqueta”.

```
import javax.swing.*;  
  
public class Ejemplo33fp [extends JFrame] {  
  
    public Ejemplo33fp() {  
        setSize(300, 200);  
        setTitle("Ventana de ejemplo");  
        createContents();  
        setVisible(true);  
    }  
  
    private void createContents() {  
        JLabel etiqueta = new JLabel("Esta es una etiqueta");  
        add(etiqueta);  
    }  
  
    public static void main(String[] args) {  
        Ejemplo33fp primeraVentana = new Ejemplo33fp();  
    }  
}
```

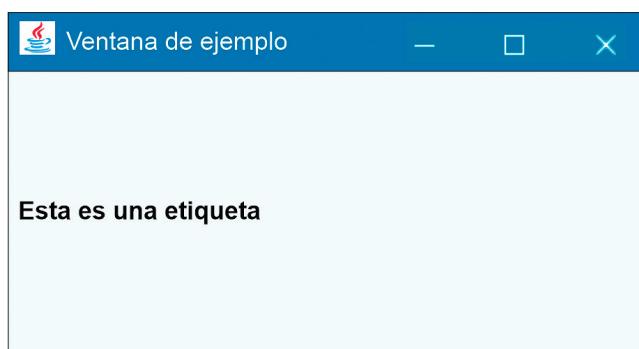


Figura 64. Resultado de la ejecución del programa

6.4. La clase JFrame

JFrame debería ser usada como la superclase para la mayoría de las aplicaciones GUI, así las ventanas a crear deberían heredar de dicha clase. Ver ejemplo anterior.

Pertenece al paquete javax.swing.

Es llamada una clase contenedora porque:

- Contiene otros componentes como: etiquetas, botones, menús, etc.
- Es derivada de la clase Container

Implementa todas las características de una clase estándar como:

- Un borde, barra de título, botones de control, capacidad de redimensionar la ventana, etc.

Principales métodos:

- `setTitle`: Muestra un título en la ventana.
- `setSize`: Establece el ancho y alto de la ventana en píxeles.
- `setLayout (new FlowLayout ())`: Asigna un manejador de layout específico a la ventana. Este manejador determina la posición de los componentes.
- `setDefaultCloseOperation(EXIT_ON_CLOSE)`: Habilita al botón X para que trabaje apropiadamente.
- `add`: Añade un componente especificado a la actual ventana.
 - Una vez añadido, permanece en la ventana durante toda la ejecución del programa.
 - En el ejemplo, aunque el label se define localmente en el método `createContents`, permanecerá con la ventana después que el método finalice.
- `setVisible(true)`: Muestra la ventana. Debemos asegurarnos de llamar a este método después de añadir todos los componentes de la ventana.
- `setVisible(false)`: Oculta la ventana.

6.5. Componentes de Java

Java tiene una gran cantidad de componentes, los principales son:

- JLabel, JTextField, JButton
- JCheckBox, JRadioButton, JComboBox, JList, JTextArea
- JMenuBar, JMenu, JMenuItem

Todos están en el paquete javax.swing, así que debemos importarlo antes de usarlos.

Todas las clases de estos componentes son descendientes de la clase JComponent que soporta varias características heredables.

Contiene métodos que manejan características de los componentes, tales como:

- Colores de foreground y background.
- Fuente de texto.

- Apariencia del borde.
- Tool tips.
- Foco.

6.6. Componente JLabel

Representa a una etiqueta o label.

Descripción:

- JLabel es un componente de sólo lectura.
- El usuario puede leer el texto de una etiqueta, pero no puede cambiarlo.
- JLabel es un componente de una simple línea, no acepta saltos.

¿Cómo implementar una etiqueta?:

- 1) Instanciar un objeto JLabel:

```
JLabel <variableReferencia> = new JLabel(<textoOpcional>);
```

- 2) Añadir el objeto JLabel a la ventana:

```
add(<variableReferencia>);
```

- 3) La clase JLabel está en el paquete javax.swing

Principales métodos de JLabel:

- 1) public String getText()
Retorna el texto de la etiqueta.

- 2) public void setText(String text)
Establece el texto de la etiqueta.

6.7. Componente JTextField

Representa un cuadro de texto.

Descripción:

- El usuario puede ingresar texto en él, no es de sólo lectura.

¿Cómo implementar un cuadro de texto?:

- 1) Crear un objeto JTextField con el constructor de JTextField:

```
JTextField <variableReferencia>= new JTextField(<textoOpc>, <anchoOpc>);
```

- 2) Añadir el objeto JTextField a la ventana:

```
add(<variableReferencia>);
```

- 3) La clase JTextField está en el paquete javax.swing

Principales métodos de JTextField:

- 1) public String getText()
Retorna el contenido del cuadro de texto.

- 2) public void setText(String text)
Establece el contenido del cuadro de texto.

- 3) `public void setEditable(boolean flag)`
Hace al cuadro de texto editable o no editable.
- 4) `public void setVisible(boolean flag)`
Hace al cuadro de texto visible o no visible.
- 5) `public void addActionListener(ActionListener listener)`
Añade un listener al cuadro de texto.

6.8. Listener de componentes

Cuando el usuario interactúa con un componente, tal como al hacer click en botón o presionar ENTER en un cuadro de texto, el componente dispara un evento.

Si el componente tiene un listener registrado o atachado a él, el evento disparado es oido por el listener y manejado por él.

El listener maneja el evento ejecutando su método `actionPerformed`.

¿Cómo implementar un listener para un cuadro de texto?

- 1) Definir una clase que implemente a la interface `ActionListener`.
- 2) Incluir un manejador de evento `actionPerformed` en la clase `listener`.

```
private class Listener implements ActionListener{
    public void actionPerformed(ActionEvent e) {
        <hacer algo>
    }
}
```

- 3) Registrar el objeto `listener` al cuadro de texto. En el ejemplo anterior:

```
nameBox.addActionListener(new Listener());
```

- 4) Es importante importar el paquete `java.awt.event`
El manejo de eventos requiere el uso de la interface `ActionListener` y de la clase `ActionEvent`. Los cuáles están en el paquete `java.awt.event`.

Notar que aunque el parámetro del manejador de evento `ActionEvent` e aún no es usado, igual debemos definirlo, ya que la JVM lo va a buscar.

EJEMPLO:

Crear una ventana con el siguiente diseño, que pida un nombre y al hacer ENTER, muestre un saludo.

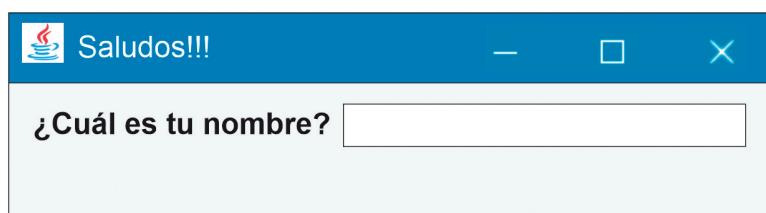


Figura 65. Resultado de la ejecución del programa

Ingrses Juancito en el JTextField y ENTER

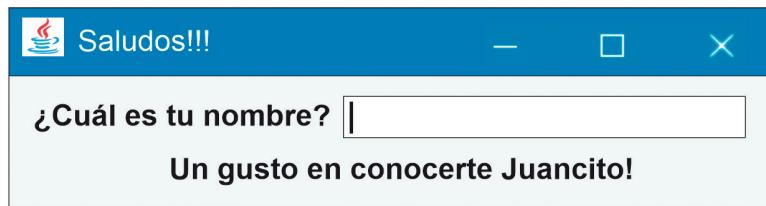


Figura 66. Resultado al ingresar Juancito y hacer ENTER

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Ejemplo34fp extends JFrame{
    private static final int ANCHO = 325;
    private static final int ALTO = 100;
    private JTextField nameBox; //ahí se escribirá nombre
    private JLabel saludo; //mostrará el saludo

    public Ejemplo34fp() {
        setTitle("Saludos!!!");
        setSize(ANCHO, ALTO);
        setLayout(new FlowLayout());
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        createContents();
        setVisible(true);
    }

    //Crea los componentes y los añade a la ventana
    private void createContents() {
        JLabel nombre = new JLabel("¿Cuál es tu nombre?");
        nameBox = new JTextField(15);
        saludo = new JLabel();
        add(nombre);
        add(nameBox);
        add(saludo);
        nameBox.addActionListener(new Listener());
    }

    // Clase interna para manejar evento
    private class Listener implements ActionListener {

        public void actionPerformed(ActionEvent e) {
            String mensaje;
            mensaje = "Un gusto conocerte, " + nameBox.getText() + "!";
            nameBox.setText("");
            saludo.setText(mensaje);
        }
    }

    public static void main(String[] args) {
        new Ejemplo34fp();
    }
}

```

6.9. Clase Interface

Recordar que una clase `interface` es como una clase en la que los métodos son todos vacíos.

Si una clase implementa a una clase `interface`, el compilador requiere que la nueva clase sobrescriba los métodos para todos los métodos de la `interface`.

Puede ser usada como una plantilla/patrón para crear clases que caen en cierta categoría.

¿Cuál es la idea de la interface `ActionListener`?

- Todos los listeners de los componentes serán similares y entendibles.
- Y así, implementarán el método `actionPerformed` y asegurará el manejo de los eventos apropiadamente.

6.10. Clases interiores (Inner classes)

Si una clase es limitada en su alcance, tal como que es sólo necesitada por otra clase, conviene definir la clase como una clase interior (definir la clase dentro de la otra clase).

Dado que un `listener` es usualmente limitado a escuchar a sólo una clase, los listeners son frecuentemente implementados como clases interiores.

Incluso, por encapsulación deberíamos ocultar la clase interior declarándola como `private`.

Una clase interior puede acceder directamente a los atributos de su clase envolvente.

Normalmente, los `listener` necesitan acceder a los atributos de su clase envolvente, entonces esto es un gran beneficio.

En el ejemplo anterior creamos la clase `Listener` como clase interior de la clase envolvente `Ejemplo34fp`.

6.11. Clases interiores anónimas

El objetivo de los objetos anónimos es evitar dar una variable de referencia (un nombre) a un objeto que sólo se necesitará usar 1 vez.

Cuando usamos una clase interior anónima es para evitar saturar el código con el nombre de una clase cuando dicha clase sólo se usará 1 vez.

En este caso, si una clase `listener` en particular escucha a sólo 1 objeto, entonces se utilizará sólo 1 vez como argumento de la llamada a `addActionListener`. En tal caso podríamos usar una clase interior anónima.

EJEMPLO:

Cambiar el código del ejemplo anterior usando una clase interior anónima.

```

private void createContents() {
    JLabel nombre = new JLabel("¿Cual es tu nombre?");
    nameBox = new JTextField(15);
    saludo = new JLabel();
    add(nombre);
    add(nameBox);
    add(saludo);
    nameBox.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            String mensaje;
            mensaje = "Un gusto conocerte, "+nameBox.getText()+"!";
            nameBox.setText("");
            saludo.setText(mensaje);
        }
    });
}

```

6.12. Componente JButton

Representa un botón, componente fundamental en cualquier interface gráfica.

Descripción:

- Un componente botón actúa como un botón del mundo real, cuando el usuario lo presiona algo pasa.

¿Cómo implementar un botón?:

- 1) Crear un objeto JButton con un constructor JButton:

```
JButton <variableReferencia> = new JButton (<textoBoton>);
```

- 2) Añadir el objeto JButton a la ventana:

```
add(<variableReferencia>);
```

- 3) La clase JButton está en el paquete javax.swing

- 4) Implementar una clase interior que incluye el método manejador de evento actionPerformed:

```

private class [Listener] implements ActionListener{
    public void actionPerformed(ActionEvent e) {
        <hacer algo>
    }
}

```

- 5) Registrar el objeto listener al botón:

```
<variableReferencia>.addActionListener(new [Listener]());
```

Principales métodos de JButton:

- 1) **public String getText()**
Retorna el texto del botón.

- 2) **public void setText(String text)**
Asigna el texto al botón.

- 3) **public void setVisible(boolean flag)**
Hace que el botón sea visible o invisible.

- 4) public void addActionListener(ActionListener listener)
 Añade un listener al botón para que escuche cuando dicho botón sea clickeado.

EJEMPLO:

Calcular el factorial de un número. Usar el siguiente diseño que debe funcionar con ENTER en cuadro de texto y con click en el botón Factorial. Mostrar "indefinido" si el número no tiene factorial.

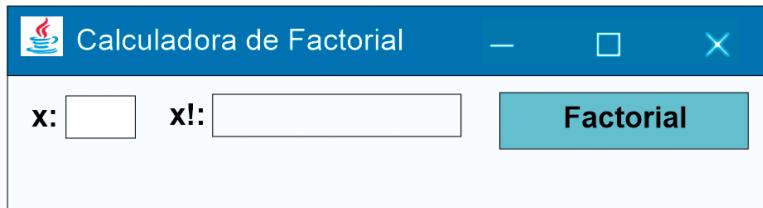


Figura 67. Resultado de la ejecución del programa

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Ejemplo35fp extends JFrame {

    private static final int ANCHO = 300;
    private static final int ALTO = 100;
    private JTextField xBox; //numero
    private JTextField xfBox; //factorial

    public Ejemplo35fp() {
        setTitle("Calculadora de Factoriales");
        setSize(ANCHO, ALTO);
        setLayout(new FlowLayout());
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        createContents();
        setVisible(true);
    }

    private void createContents() {
        JLabel xLabel = new JLabel("x:");
        JLabel xfLabel = new JLabel("x!:");
        JButton btn = new JButton("Factorial");
        Listener listener = new Listener();

        xBox = new JTextField(2);
        xfBox = new JTextField(10);
        xfBox.setEditable(false);
        add(xLabel);
        add(xBox);
        add(xfLabel);
        add(xfBox);
        add(btn);
        xBox.addActionListener(listener);
        btn.addActionListener(listener);
    }
}
```

```

//Clase interna
private class Listener implements ActionListener {

    public void actionPerformed(ActionEvent e) {
        int x;           //numero
        int xf;          //factorial de numero

        try {
            x = Integer.parseInt(xBox.getText());
        } catch (NumberFormatException nfe) {
            x = -1;          //indica invalido x
        }
        if (x < 0)
            xfBox.setText("indefinido");
        else {
            if (x == 0 || x == 1)
                xf = 1;
            else {
                xf = 1;
                for (int i = 2; i <= x; i++)
                    xf *= i;
            }
            xfBox.setText(Integer.toString(xf));
        }
    }
}

public static void main(String[] args) {
    new Ejemplo35fp();
}
}

```

6.13. Cuadro de diálogo JOptionPane

Un cuadro de diálogo es una ventana simple que contiene un mensaje y lo crearemos usando la clase `JOptionPane`.

Para generar un cuadro de diálogo de salida simple, usaremos el método `showMessageDialog`:

SINTAXIS:

```
JOptionPane.showMessageDialog(<container>, <mensaje>);
```

`showMessageDialog` es un método de clase, así que usaremos la notación `.dot`.

El primer argumento `container` indica la posición del cuadro de diálogo. Si el argumento es `null`, aparece centrado en toda la pantalla.

El primer argumento especifica el contenedor que rodeará al cuadro de diálogo. El cuadro de diálogo se mostrará en el centro del contenedor.

El segundo argumento `mensaje` indica el mensaje a mostrar en el cuadro de diálogo.

La clase `JOptionPane` se encuentra en el paquete `javax.swing`

EJEMPLO:

Crear un cuadro de diálogo que muestre el mensaje: "Presionar el botón factorial para realizar cálculo" y que aparezca centrado en la pantalla.

```
JOptionPane.showMessageDialog(null, "Presionar el botón factorial para realizar cálculo");
```



Figura 68. Ventana generada con JOptionPane

6.14. Clase Color

La mayoría de los componentes GUI están compuestos de 2 colores: **foreground** color es el color del texto y **background** color es el color del área detrás del texto.

Para gestionarlos usaremos la clase `Color`.

EJEMPLO:

Crear un botón rojo con texto blanco.

```
 JButton btn = new JButton("Presioname");
btn.setBackground(Color.RED);
btn.setForeground(Color.WHITE);
```

Usaremos `setBackground` y `setForeground` como métodos mutadores y como métodos accesores usaremos `getBackground` y `getForeground`.

EJEMPLO:

Crear un cuadro de texto y guardar en 2 variables de tipo `Color` su color de fondo y de texto.

```
JTextField nameBox = new JTextField();
Color originalBackground = nameBox.getBackground();
Color originalForeground = nameBox.getForeground();
```

En la siguiente tabla se puede ver la lista de colores y sus respectivas constantes nombradas en Java.

Contante de Color	Color relacionado
Color.BLACK	Negro
Color.GREEN	Verde
Color.RED	Rojo
Color.BLUE	Azul
Color.LIGHT_GRAY	Plomo claro

Color.WHITE	Blanco
Color.YELLOW	Amarillo
Color.CYAN	Celeste
Color.MAGENTA	Fucsia
Color.DARK_GRAY	Plomo oscuro
Color.ORANGE	Naranja
Color.GRAY	Plomo
Color.PINK	Rosado

Tabla 5. Principales constantes de Color

La clase `Color` está en el paquete `java.awt`.

Para obtener un color que no está en la lista de colores de constantes nombradas, debemos instanciar un objeto `Color` con una mezcla de rojo, verde y azul (RGB).

SINTAXIS:

```
new Color(<rojo 0-255>, <verde 0-255>, <azul 0-255>)
```

Donde 0 es sin color, 255 es el máximo color.

EJEMPLO:

Para establecer un color de fondo morado suave a un botón:

```
button.setBackground(new Color(128, 0, 128));
```

Si lo que deseamos es establecer el background color de una ventana `JFrame`, primero debemos conseguir el panel de contenido del `JFrame` y después aplicarle el background color.

Usaremos el método `getContentPane` para conseguir dicho panel.

EJEMPLO:

El siguiente código conseguirá el panel del `JFrame` y lo pondrá de color amarillo.

```
getContentPane().setBackground(Color.YELLOW);
```

EJEMPLO:

Crear una GUI con el siguiente diseño. Cuando presione el botón `Parar`, el fondo de la ventana se pondrá rojo y cuando presione el botón `Avanzar`, el fondo se pondrá verde.



Figura 69. Interface de la aplicación a crear

```

import javax.swing.*;      //para JFrame & JButton
import java.awt.*;        //para FlowLayout, Color, & Container
import java.awt.event.*;  //para ActionListener & ActionEvent

public class Ejemplo36fp extends JFrame{

    private static final int ANCHO = 300;
    private static final int ALTO = 100;
    private JButton altoBoton;
    private JButton avanzaBoton;

    public Ejemplo36fp() {
        setTitle("Elegir color de fondo");
        setSize(ANCHO, ALTO);
        setLayout(new FlowLayout());
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        createContents();
        setVisible(true);
    }

    private void createContents() {
        altoBoton = new JButton("Parar");
        altoBoton.setBackground(Color.RED);
        altoBoton.addActionListener(new ButtonListener());
        add(altoBoton);

        avanzaBoton = new JButton("Avanzar");
        avanzaBoton.setBackground(new Color(0, 255, 0));
        avanzaBoton.addActionListener(new ButtonListener());
        add(avanzaBoton);
    }
    //Clase Interna
    private class ButtonListener implements ActionListener {

        public void actionPerformed(ActionEvent e) {
            Container contentPane = getContentPane();

            if (e.getSource() == altoBoton)
                contentPane.setBackground(Color.RED);
            //sin crear contentPane sería
            //getContentPane().setBackground(Color.RED);
            else
                contentPane.setBackground(Color.GREEN);
            //sin crear contentPane sería
            //getContentPane().setBackground(Color.GREEN);
        }
    }

    public static void main(String[] args) {
        new Ejemplo36fp();
    }
}

```

6.15. Otros Componentes

- 1) **JTextArea**: lo usaremos para mostrar múltiples líneas de texto.

SINTAXIS:

```
JTextArea <referenciaJTextArea> = new JTextArea(<texto>);
```

Principales métodos:

```
public String getText()
```

Retorna el texto del área de texto.

```
public void setText(String text)
```

Establece el texto del área de texto.

```
public void setEditable(boolean flag)
```

Hace el área de texto editable o no editable.

```
public void setLineWrap(boolean flag)
```

Permite que muestre en varias líneas o no.

```
public void setWrapStyleWord(boolean flag)
```

Permite mostrar palabras completas para el salto de línea. false, corta las palabras.

- 2) **JCheckBox**: lo usaremos para tener un cuadro que permita seleccionar o deseleccionar alguna opción, tipo interruptor. Cuando tenemos varios checkbox se pueden tener varios de ellos seleccionados a la vez.

SINTAXIS:

```
JCheckBox <referencia> = new JCheckBox(<label>, <selec>);
```

Principales métodos:

```
public boolean isSelected()
```

Retorna true si el cuadrito está seleccionado, falso si no.

```
public String setVisible(boolean flag)
```

Hace al checkbox visible o no.

```
public void setSelected(boolean flag)
```

Hace al checkbox seleccionado o no.

```
public void setEnabled(boolean flag)
```

Hace al checkbox habilitado o deshabilitado.

```
public void addActionListener(ActionListener listener)
```

Añade un listener al checkbox.

- 3) **JRadioButton**: lo usaremos para seleccionar una entre varias opciones.

SINTAXIS:

```
JRadioButton <referencia> = new JRadioButton(<label>, <selec>);
```

Para habilitar la funcionalidad de “sólo un botón seleccionado a la vez”, debemos crear un objeto `ButtonGroup` y añadir botones individuales a él (agruparlos):

```
ButtonGroup <referenciaButtonGroup> = new ButtonGroup();  
<referenciaButtonGroup>.add(<primerBotónDelGrupo>);  
...  
<referenciaButtonGroup>.add(<últimoBotónDelGrupo>);
```

Principales métodos:

```
public boolean isSelected()  
    Retorna true si el radiobutton está seleccionado, falso si no.
```

```
public String setVisible(boolean flag)  
    Hace al radiobutton visible o no.
```

```
public void setSelected(boolean flag)  
    Hace al radiobutton seleccionado o no.
```

```
public void setEnabled(boolean flag)  
    Hace al radiobutton habilitado o deshabilitado.
```

```
public void addActionListener(ActionListener listener)  
    Añade un listener al radiobutton.
```

- 4) **JComboBox**: lo usaremos para seleccionar un ítem de un conjunto de ítems, también se les llama listas drop-down o desplegables.

SINTAXIS:

```
JComboBox <referencia> = new JComboBox(<arregloItems>);
```

EJEMPLO:

Crear un **JComboBox** con los días laborables de la semana como ítems.

```
private String[] dias={"Lunes", "Martes", "Miercoles",  
"Jueves", "Viernes"};  
JComboBox cajaDias = new JComboBox(dias);
```

Principales métodos:

```
public String setVisible(boolean flag)  
    Hace el combobox visible o no.
```

```
public void setEditable(boolean flag)  
    Hace la parte del cuadro de texto del combobox editable o no.
```

```
public Object getSelectedItem()  
    Retorna el ítem seleccionado actualmente.
```

```
public void setSelectedItem(Object item)  
    Cambia el ítem actualmente seleccionado, reemplazándolo por el ítem enviado  
    como argumento.
```

```
public void addActionListener(ActionListener listener)  
    Añade un listener al combobox.
```

EJERCICIOS PROPUESTOS

1. Investigar sobre el uso de Layout managers: **FlowLayout**, **BorderLayout** y **GridLayout**. Dar ejemplos.
2. Profundizar sobre el uso de los componentes: **JTextArea**, **JCheckBox**, **JRadioButton**, **JComboBox** y **JPanel**. Dar ejemplos.

CAPÍTULO 7

ARCHIVOS

OBJETIVOS:

- Comprender cómo escribir y leer de archivos de texto
- Comprender cómo escribir y leer de archivos binarios
- Diferenciar los archivos de texto de los archivos binarios
- Comprender cómo escribir y leer de archivos de objetos
- Solucionar problemas aplicando archivos de texto, binarios y de objetos

7.1. Fundamentos de Entrada/Salida (E/S)

Hasta aquí toda E/S de datos (I/O) se realizó por medio del teclado (entrada) y se dirigió a la ventana de consola de la computadora (salida). Dichas E/S son temporales y no permanentes.

Así, los valores de las variables que ingresamos en nuestros programas se almacenan temporalmente en la memoria RAM. Muchas veces es necesario que dichos datos se hagan permanentes, que permanezcan aunque la computadora se apague. Para lo cual utilizaremos archivos (files) por medio de **streams** (flujo secuencial de datos).

El beneficio de **leer** entradas desde un archivo es que permite que las entradas sean reusadas indefinidamente sin tener que reingresar los datos por el teclado.

El beneficio de **escribir** salidas en un archivo es que permite un almacenamiento permanente, permitiendo que las salidas sean revisadas una y otra vez sin tener que volver a ejecutar el programa que las generó. También permite que un programa pueda enlazar la salida generada por otro programa y lo utilice como su entrada.

Para manipular archivos, necesitamos usar varias clases predefinidas de la API de Java. Recordar que la API de Java está organizada como una jerarquía de paquetes donde cada paquete contiene un grupo de clases. El paquete que contiene las clases para E/S es `java.io`.

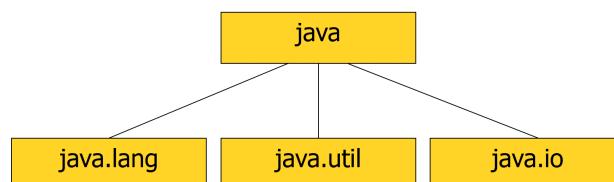


Figura 70. Ubicación de paquete `java.io` para E/S

7.2. Clases para E/S de archivos

Tenemos varias clases para tratar la E/S de archivos.

A veces existe un cierto traslape entre tales clases, algunas clases manejan el mismo tipo de operaciones que otras. Pueden haber varias formas de hacer la misma tarea.

Existen diferentes tipos de archivos y para cada uno usaremos las clases más prácticas para su respectiva E/S:

1. Archivos de Texto

- Cuando escribimos texto en un archivo, almacenamos el texto como caracteres donde cada carácter es representado por su código ASCII. Un archivo de texto es independiente de cualquier sistema operativo o aplicación.
- Si deseamos **escribir** texto en un archivo usaremos la clase `PrintWriter`.
- Si deseamos **leer** texto desde un archivo, usaremos las clases `Scanner` y `FileReader`.

2. Archivos Binarios

- Cuando escribimos datos nativos de Java (no como texto) en un archivo, almacenamos en su formato nativo **bytecode**, no como caracteres ASCII.
- Si deseamos **escribir** datos nativos (no como texto) en un archivo usaremos las clases `DataOutputStream` y `FileOutputStream`.
- Si deseamos **leer** datos nativos (no como texto) desde un archivo, usaremos las clases `DataInputStream` y `FileInputStream`.

3. Archivos de Objetos

- Cuando escribimos un objeto en un archivo, almacenamos los atributos del objeto en su formato nativo **bytecode**, no como caracteres ASCII.
- Si deseamos **escribir** objetos en un archivo, usaremos las clases `ObjectOutputStream` y `FileOutputStream`.
- Si deseamos **leer** objetos desde un archivo, usaremos las clases `ObjectInputStream` y `FileInputStream`.



Figura 71. Principales clases utilizadas para E/S según el tipo de archivo

7.3. Operaciones básicas para E/S de archivos

Existen 4 pasos básicos para realizar la E/S de archivos:

1. Importar las clases necesarias.
 - Para entrada y salida importar el paquete `java.io`.
 - Para entrada, además importar la clase `java.util.Scanner`.
2. Abrir el archivo requerido por medio de la instanciación de la clase o clases apropiadas según el **tipo del archivo** a abrir. Se puede abrir un archivo para lectura o para escritura.
3. Escribir en o leer desde el archivo abierto, llamando a los métodos apropiados.
 - Con archivo abierto para salida, sólo las operaciones de escritura son permitidas.
 - Con archivos abierto para entrada, sólo las operaciones de lectura son permitidas.
4. Cerrar el archivo con el método `close`.

7.4. Salida en archivos de texto

Para abrir un archivo de texto para salida:

- 1) Instanciar la clase `PrintWriter`.

```
PrintWriter <variableReferencia> = new PrintWriter(<filename>);
```

- El constructor de `PrintWriter` lanza una excepción `FileNotFoundException`.
- Para manejar la excepción, rodear la llamada al constructor con un bloque `try`, y añadir un bloque `catch` para la excepción `FileNotFoundException`.

- 2) Para escribir en el archivo de texto abierto.

Usar los métodos `print` y `println` del objeto instanciado de `PrintWriter`.

- 3) Para cerrar un archivo de texto abierto.

Llamar al método `close` del objeto `PrintWriter`.

```
<variableReferencia>.close();
```

EJEMPLO:

En el archivo `veamos.txt` escribir el texto `hola`.

Observar el uso del manejo de excepciones.

```
import java.io.*;  
  
public class Ejemplo37fp {  
  
    public static void main(String[] args) {
```

```
try {
    PrintWriter fileOut = new PrintWriter("veamos.txt");
    fileOut.println("hola");
    fileOut.close();
    System.out.println("Archivo guardado");
} catch (FileNotFoundException e) {
    System.out.println("Error: " + e.getMessage());
}
}
```

EJEMPLO:

En un archivo, cuyo nombre ingresemos, escribir el texto Bienvenidos a OO.

Comprobar que **sobrescribe** si había algún texto en dicho archivo.

```
import java.util.*;
import java.io.*;

public class Ejemplo38fp {

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        PrintWriter fileOut;
        String text = "Bienvenidos a OO";

        try {
            System.out.print("Enter filename: ");
            fileOut = new PrintWriter(scan.nextLine());
            fileOut.println(text);
            fileOut.close();
        } catch (FileNotFoundException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

Por default, cuando abrimos un archivo con `PrintWriter`, el contenido del archivo es borrado.

El primer `print` causará que la salida se ubique al comienzo del archivo vacío.

Si no queremos borrar el contenido debemos abrir el archivo en modo de añadir (append mode) insertando un objeto `FileWriter` en la llamada al constructor de `PrintWriter`. Esto causa que los `print` inserten el texto al final del archivo.

SINTAXIS:

```
PrintWriter <variableReferencia>;
<variableReferencia> = new PrintWriter(new FileWriter(<nombreArchivo>,true));
```

EJEMPLO:

En un archivo, cuyo nombre ingresemos, escribir el texto este es un texto extra.

Comprobar que el programa añade dicho texto a continuación del texto existente en el archivo indicado.

```
import java.util.*;
import java.io.*;

public class Ejemplo39fp {

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        PrintWriter fileOut;
        String text = "este es un texto extra";

        try {
            System.out.print("Enter filename: ");
            fileOut = new PrintWriter(new FileWriter(scan.nextLine(), true));
            fileOut.println(text);
            fileOut.close();
        } catch (IOException e) {
            System.out.println("IO: " + e.getMessage());
        }
    }
}
```

7.5. Entrada desde archivos de texto

Para abrir un archivo de texto para entrada:

- 1) Instanciar las clases `Scanner` y `FileReader`.

```
Scanner <variableReferencia>;
<variableReferencia> = new Scanner(new FileReader(<nombreArchivo>));
```

El constructor de `FileReader` lanza una `FileNotFoundException`. Para manejar la excepción, rodear la llamada al constructor con un bloque `try`, y añadir un bloque `catch` para la excepción `FileNotFoundException`.

- 2) Para leer desde un archivo de texto abierto.
 - Llamar a los métodos `next` del objeto `Scanner`.
 - Usaremos los métodos `next`: `nextLine` , `nextInt`, `nextDouble`, etc.
- 3) Para cerrar el archivo de texto abierto.
 - Llamar al método `close` del objeto `Scanner`.

```
<variableReferencia>.close();
```

EJEMPLO:

En el archivo `veamos.txt` leemos su primera línea, el texto `hola`. El archivo debe estar en la misma carpeta del proyecto para poder leerlo.

```
import java.util.*;
import java.io.*;
```

```
public class Ejemplo40fp {  
  
    public static void main(String[] args) {  
        String line;  
  
        try {  
            Scanner fileIn = new Scanner(new FileReader("veamos.txt"));  
            line = fileIn.nextLine();  
            System.out.println(line);  
            fileIn.close();  
        } catch (FileNotFoundException e) {  
            System.out.println("Error: " + e.getMessage());  
        }  
    }  
}
```

EJEMPLO:

En el archivo `veamos.txt` leemos su texto línea por línea. El archivo debe estar en la misma carpeta del proyecto para poder leerlo. Observar el uso del método `hasNextLine`.

```
import java.util.*;  
import java.io.*;  
  
public class Ejemplo41fp {  
  
    public static void main(String[] args) {  
        String line;  
  
        try {  
            Scanner fileIn = new Scanner(new FileReader("veamos.txt"));  
            while (fileIn.hasNextLine()) {  
                line = fileIn.nextLine();  
                System.out.println(line);  
            }  
            fileIn.close();  
        } catch (FileNotFoundException e) {  
            System.out.println("Error: " + e.getMessage());  
        }  
    }  
}
```

Cuando leemos texto utilizando un objeto `Scanner` (con sus métodos `nextLine`, `next`, `nextInt`, etc.), leemos **secuencialmente** de inicio a fin.

La primera lectura se realiza al comienzo del archivo y así lee hacia el final del archivo.

Si se intenta leer más allá del fin del archivo el programa lanza la excepción `NoSuchElementException`.

Para prevenir dicho problema, usamos el método `hasNextLine` de `Scanner` antes de la lectura desde el archivo.

El método `hasNextLine` retorna `true` si es seguro leer desde el archivo. Retorna `false` si no hay más líneas que leer.

```
while (fileIn.hasNextLine())
    System.out.println(fileIn.nextLine());
```

También podemos usar el método `hasNext` que va leyendo por tokens (palabras) y no por líneas. Así, retorna `true` si es seguro leer un token desde el archivo y retorna `false` si no hay más tokens que leer.

7.6. Formato de texto vs Formato binario

Java puede manejar 2 tipos de formatos de archivos:

1) Formato de archivo de texto.

Los datos son almacenados usando valores ASCII 8-bit.

Pueden ser vistos usando editores de texto y pueden ser leídos por programas escritos en cualquier lenguaje de programación (no sólo Java).

Para facilitar la lectura y escritura de datos línea por línea, los datos del archivo de texto están organizados usando caracteres de nueva línea.

Cuando escribimos en un archivo de texto, Java usa el método `println` para especificar que una nueva línea se inserta al final del archivo.

Cuando se lee desde un archivo, Java usa el método `nextLine` para especificar que una línea completa se leerá. También se pueden usar otros métodos `next`.

2) Formato de archivo binario.

Los datos se almacenan usando el esquema de almacenamiento nativo de Java o bytecode (almacena datos en el archivo tal como almacena en la memoria RAM).

Estos archivos no facilitan la lectura y escritura línea por línea, preferentemente no usan los métodos `println` ni `nextLine`.

En su lugar usan los métodos: `writeChar`, `readChar`, `writeInt`, `readInt`, `writeDouble`, `readDouble`, etc. para escribir y leer variables individuales de tipo primitivo.

Beneficios del formato de texto:

- Fácil de ver el archivo, cualquier editor de texto lo puede visualizar.
- Puede ser leído por programas escritos en otros lenguajes de programación.

Beneficios del formato binario:

- El formato de texto sólo puede manejar los valores ASCII, pero el formato binario puede manejar TODOS los caracteres Unicode.
- Permite crear programas que corren más rápido, ya que cuando un programa escribe o lee de un archivo binario, los datos no necesitan ser convertidos. En cambio cuando un programa escribe o lee de un archivo de texto, los datos tienen que ser convertidos entre el formato nativo de Java y el formato ASCII.
- Puede almacenar objetos complejos.

7.7. Salida en archivos binarios

Para abrir un archivo binario como salida de datos primitivos, debemos instanciar un objeto `FileOutputStream`.

Para transformar tipos de datos primitivos en bytes, debemos instanciar un objeto `DataOutputStream`.

EJEMPLO:

```
DataStream fileOut;
...
fileOut=new DataOutputStream(new FileOutputStream(nombreArch, true));
...
```

EJEMPLO:

Escribir en un archivo binario cuyo nombre ingresemos, n números enteros a partir de 0.

```
import java.io.*;
import java.util.Scanner;

public class Ejemplo42fp {

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        DataOutputStream fileOut;
        int numRegistros;

        try {
            System.out.print("Ingrese el nombre del archivo a guardar: ");
            fileOut = new DataOutputStream(new FileOutputStream(scan.nextLine()));

            System.out.print("Ingrese la cantidad de enteros a guardar: ");
            numRegistros = scan.nextInt();
            for (int i = 0; i < numRegistros; i++)
                fileOut.writeInt(i);

            fileOut.close();
        } catch (IOException e) {
            e.getMessage();
        }
    }
}
```

7.8. Entrada desde archivos binarios

Para abrir un archivo binario para entrada de datos primitivos, debemos instanciar un objeto `FileInputStream`.

Para transformar los tipos de datos primitivos en bytes, debemos instanciar un objeto `DataInputStream`.

EJEMPLO:

```
DataInputStream fileIn;  
...  
fileIn = new DataInputStream(new FileInputStream(filename));  
...
```

Métodos de DataInputStream a utilizar:

```
int readInt()    char readChar()        double readDouble()
```

EJEMPLO:

Suponer que se tiene un arreglo unidimensional doubleValues, llenarlo con datos del archivo binario creado previamente, pero que sean multiplicados por 2. Usar el método readDouble de DataInputStream e imprimir los elementos del arreglo en pantalla.

```
DataInputStream fileIn= new DataInputStream(new FileInputStream("prueba.dat"));  
  
for (int i=0; i<doubleValues.length; i++){  
    doubleValues[i] = fileIn.readDouble()*2;  
    System.out.println(doubleValues[i]);  
}
```

7.9. E/S en un archivo de objetos

En la POO, la mayoría de los datos pertenecen a un objeto (valores de los atributos del objeto) y la clase, a partir de la que se crean los objetos, es muchas veces especificada por el usuario.

Entonces, no hay un formato universal para almacenar objetos en un archivo, y cada atributo debe ser almacenado separadamente usando un formato apropiado para el tipo de objeto.

Esto genera que escribir y leer datos de objetos en y desde un archivo sea algo tedioso. La solución sería automatizar este proceso usando el servicio de serialización de Java y permitir trabajar con el objeto completo (todos sus atributos).

Para conseguir utilizar este servicio, se debe aumentar el siguiente código a la cabecera de cualquier clase que deseemos que use el servicio:

```
implements Serializable
```

La JVM manejará todos los detalles y se encargará de la serialización (escribir el objeto en el archivo) y la deserialización (leer el objeto desde archivo).

EJEMPLO:

La clase serializable TestObject debería ser definida:

```
public class TestObject implements Serializable { ... }
```

Para salida (escribir objetos en el archivo):

```
//testObj es objeto de la clase TestObject
```

```
//filename es el nombre del archivo utilizado

ObjectOutputStream fileOut;
fileOut= new ObjectOutputStream(new FileOutputStream(filename));
fileOut.writeObject(testObj);
fileOut.close();
```

Para entrada (leer objetos desde el archivo), observar el casteo del objeto:

```
//testObj es objeto de la clase TestObject
//filename es el nombre del archivo utilizado
```

```
ObjectInputStream fileIn;
fileIn= new ObjectInputStream(new FileInputStream(filename));
testObj = (TestObject) fileIn.readObject();
fileIn.close();
```

EJEMPLO:

Almacenar 2 objetos de la clase TestObject que tenga 3 atributos: id, texto y numero de tipos int, String y double respectivamente en un archivo de objetos cuyo nombre ingresemos, luego que los lea y los imprima en pantalla.

```
import java.util.*;
import java.io.*;

public class Ejemplo43fp {

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        ObjectOutputStream fileOut;
        TestObject unObjeto = new TestObject(1, "test", 2.0);
        TestObject otroObjeto = new TestObject(2, "ver", 3.0);
        String filename;

        System.out.print("Ingresa nombre del archivo: ");
        filename = scan.nextLine();
        try {
            fileOut = new ObjectOutputStream(new FileOutputStream(filename));
            fileOut.writeObject(unObjeto);
            fileOut.writeObject(otroObjeto);

            fileOut.close();
        } catch (IOException e) {
            System.out.println("Error: " + e.getMessage());
        }

        ObjectInputStream fileIn;
        TestObject unObjetoLectura;
```

```
try {
    fileIn = new ObjectInputStream(new FileInputStream(filename));
    unObjetoLectura = (TestObject) fileIn.readObject();
    unObjetoLectura.display();
    unObjetoLectura = (TestObject) fileIn.readObject();
    unObjetoLectura.display();

    fileIn.close();
} catch (IOException e) {
    System.out.println("IO Error: " + e.getMessage());
} catch (ClassNotFoundException e) {
    System.out.println("ClassNotFoundException " + e.getMessage());
}
}

//TestObject.java
import java.io.*;

public class TestObject implements Serializable{
    private int id;
    private String texto;
    private double numero;
    public TestObject(int i, String t, double num) {
        id = i;
        texto = t;
        numero = num;
    }

    public void display() {
        System.out.print(id + "\t");
        System.out.print(texto + "\t");
        System.out.println(numero);
    }
}
```

EJERCICIOS PROPUESTOS

1. Imprimir en el archivo mensaje.txt el texto: hola amigos, como están.
2. Imprimir en el archivo numeros.txt los 10 primeros números enteros mayores que 0.
3. Imprimir en el archivo numerosAleatorios.txt los 10 lanzamientos aleatorios de un dado.
4. Programa que almacene en un arreglo 10 lanzamientos aleatorios de un dado, luego que los escriba en un archivo de texto. Usando métodos.
5. Imprimir en un archivo, cuyo nombre ingresemos por teclado, un mensaje que también ingresemos por teclado.
6. El anterior, pero abrir el archivo por segunda vez, ahora en modo añadir y añadir un segundo mensaje que también ingresemos por teclado.
7. Leer del archivo mensaje.txt el texto escrito previamente.
8. Leer del archivo numeros.txt los 10 primeros números enteros mayores que 0 que fueron almacenados.
9. Leer del archivo numerosAleatorios.txt los 10 lanzamientos aleatorios de un dado ya almacenados.
10. Programa que lea los 10 lanzamientos aleatorios de un dado ya almacenados, luego que los guarde en un arreglo. Usando métodos.
11. Programa que almacene la tabla de multiplicar de un número en un intervalo dado, en un archivo de texto y que después la lea y la muestre en pantalla.
12. Almacenar el abecedario en un archivo de texto, luego anexar en el mismo archivo la palabra "CASA", después leer todos los datos almacenados en el archivo y mostrarlos.
13. Almacenar en un archivo la lista de n estudiantes con datos: nombre, edad, estatura y luego leerlos, en archivo de texto.
14. Generar un conjunto de N enteros aleatorio en intervalo [x..y] y almacenarlos en un archivo binario, luego leerlos, ordenarlos y almacenarlos en otro archivo binario.
15. Almacenar N objetos de la clase Estudiante con atributos cui, nombre y edad, luego leerlos, ordenarlos por edad y almacenarlos debidamente ordenados en otro archivo de objetos.

CAPÍTULO 8

BASES DE DATOS

OBJETIVOS:

- Comprender los conceptos básicos de las Bases de Datos (BD)
- Entender la importancia de las Bases de Datos en la sociedad de la información y del conocimiento
- Comprender y aplicar conceptos fundamentales de modelado de BD
- Entender los fundamentos del Structured Query Language (SQL)
- Comprender cómo enlazar un programa en Java a una BD

8.1. Definiciones

1) ¿Qué es un dato?

Atributo o característica de una entidad que puede ser procesado.

2) ¿Qué es una Base de Datos (BD)?

Una colección coherente de datos relacionados.

3) ¿Qué es Minimundo o Dominio del Problema?

Parte del mundo real acerca de la cual se almacenan datos en la BD.

4) ¿Qué es un Sistema Gestor de Base de Datos (SGBD) (DBMS)?

Software que facilita la creación, administración y mantenimiento de BD.

Los más usados: MySQL, PostgreSQL, SQL Server, Oracle, MS Access, etc.

8.2. Bases de Datos

Una BD es una colección de datos relacionados y tiene las siguientes propiedades:

- Representa ciertos aspectos del mundo real (minimundo o dominio del problema). Se centra en un dominio en particular.
- Es una colección coherente de datos. Los datos se relacionan entre sí.
- Se diseña, construye y puebla con datos para un propósito específico.

8.3. Sistema Gestor de Bases de Datos (SGBD)

Es una categoría de software que tiene las siguientes funciones principales:

- Permitir a los usuarios crear BD.
- Ofrecer a los usuarios la capacidad de consultar y actualizar los datos en forma eficiente.
- Soportar el almacenamiento de cantidades voluminosas de datos, protegiéndolos y brindándoles seguridad.

- Controlar el acceso concurrente a los datos por muchos usuarios.
- Permitir la administración en general de las BD.

8.4. Anatomía de una Base de Datos

Una base de datos es la colección de 1 ó más tablas.

Una tabla es la colección de información relacionada (registros).

Un registro es la información relacionada a una persona, producto, evento, etc. en particular. Conjunto de valores para los campos que se refieren a cierta entidad.

Un campo es una parte discreta de información en un registro.

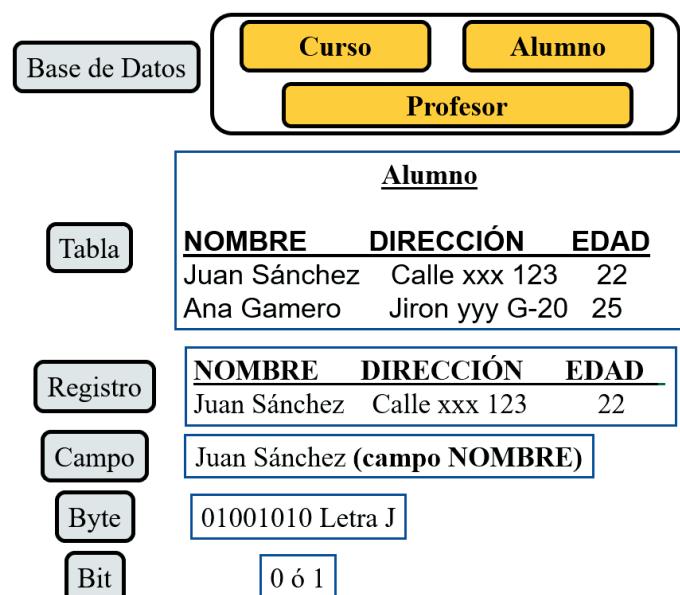


Figura 72. Esquema de la anatomía de una base de datos

Este diagrama muestra una tabla titulada 'Alumno' con los siguientes datos:

	Nombre	Direccion	Edad	EstadoCivil
1	Ana Gamero	Jiron Juan Carlos G-20	25	Soltero
2	Eli Vidal	Cono Norte 179	43	Soltero
3	Evel Castro	Huaranguillo J-15	48	Casado
4	Juan Sánchez	Calle Las Orquídeas 123	22	Soltero

Figura 73. Ejemplo de tabla: Alumno

8.5. Modelado de una base de datos

Una base datos debe ser debidamente diseñada antes de ser creada, para lo cual se debe realizar:

- Diseño Conceptual: un diagrama Entidad – Relación (ER) o equivalente.
- Diseño Lógico: un diagrama Relacional, que es la transformación de un ER.
- Diseño Físico: esquema interno de la BD en un SGBD específico, se basa en el diagrama Relacional.



Figura 74. Fases de Diseño de una BD

8.6. Caso de estudio

Se desea simular el comportamiento de una universidad cualquiera, atendiendo a la siguiente especificación:

- Cada alumno se matricula en una serie de asignaturas en las que obtendrá una calificación. En una asignatura se pueden matricular muchos alumnos.
- Cada asignatura es impartida por un único profesor y un profesor puede dar clase de muchas asignaturas. Cada profesor pertenece a un departamento.
- De los alumnos almacenaremos el dni, nombre, primer apellido, segundo apellido, calle, ciudad, provincia, teléfono, fecha de nacimiento y estado civil. El estado civil de un alumno puede ser el siguiente: S:Soltero C:Casado V:Viudo P:Separado D:Divorciado.
- De las asignaturas se guarda el código, nombre completo, número de créditos y un campo de observaciones.
- De cada profesor tenemos información del dni, titulación que posee, nombre y primer apellido.
- De los departamentos se tiene como datos su código y nombre.
- Un alumno puede matricularse en la misma asignatura varias veces.

Realizar el Modelo ER y Modelo Relacional.

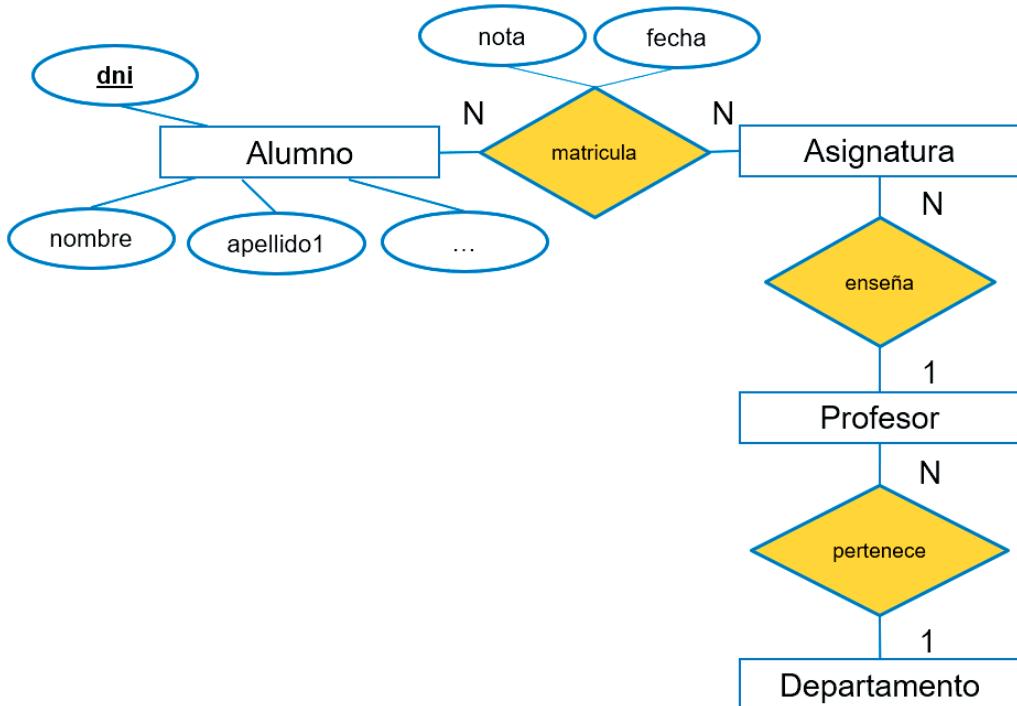


Figura 75. Modelo ER del caso de estudio (Diseño Conceptual)

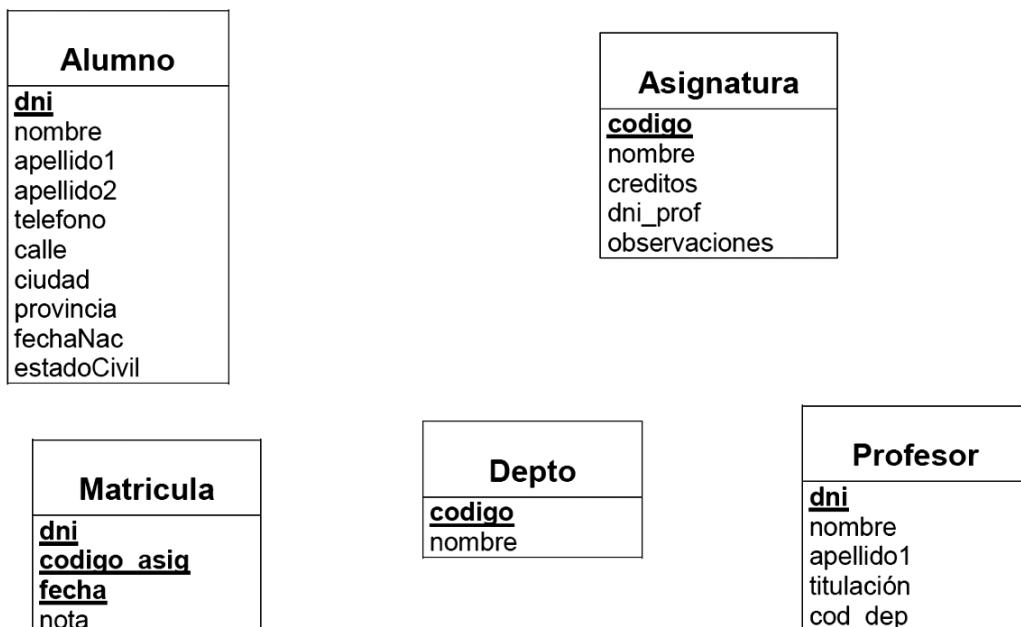


Figura 76. Modelo Relacional del caso de estudio (Diseño Lógico derivado del Modelo ER)

8.7. Lenguaje SQL

El lenguaje SQL es un lenguaje de consulta estructurado sobre las BD y está compuesto de:

SQL = DDL + DML

- 1) DDL: Lenguaje de Definición de Datos.

- Sentencias del SQL que permiten definir los objetos de la Base de Datos.
 - CREATE, DROP, ALTER.
- 2) DML: Lenguaje de Manipulación de Datos.
- Sentencias del SQL que permiten consultar y actualizar los datos de la Base de Datos.
 - SELECT, INSERT, UPDATE, DELETE.

8.8. Data Management Language (DML)

Para los objetivos del presente texto, describiremos las sentencias principales del DML.

Tenemos sentencias básicas para consultar, insertar, eliminar y actualizar.

1) La sentencia básica de consulta es SELECT.

SINTAXIS:

```
SELECT <Lista de atributos>
FROM   <Lista de tablas>
[WHERE <condiciones>]
```

Se lee: selecciona los atributos x, y, z desde la/las tabla/tablas t1, t2 donde x = a.

EJEMPLO:

Mostrar todos los datos de los alumnos:

```
SELECT * FROM Alumno
```

Mostrar nombre, apellidos y fecha de nacimiento de los alumnos:

```
SELECT nombre, apellido1, apellido2, fechaNac FROM alumno
```

Mostrar sólo los alumnos solteros:

```
SELECT * FROM alumno WHERE estadocivil='S'
```

Mostrar sólo los alumnos solteros o casados:

```
SELECT * FROM alumno WHERE estadocivil='S' or estadocivil='C'
```

Mostrar los que viven en Arequipa y son solteros:

```
SELECT * FROM alumno WHERE provincia='Arequipa' and estadocivil='S'
```

Nombre y número de créditos de las asignaturas que tienen un número de créditos comprendido entre 4 y 6:

```
SELECT nombre, creditos FROM Asignatura WHERE creditos>=4 and creditos<=6
```

```
SELECT nombre, creditos FROM Asignatura WHERE creditos between 4 and 6
```

Lista de todos los datos de departamento y profesores pertenecientes a él (2 tablas):

```
SELECT * FROM depto, profesor WHERE codigo=cod_dep
```

Lista de nombre de departamento, nombre y apellido del departamento de Estadística y Matemática Aplicada:

```
SELECT depto.nombre, dni, profesor.nombre, apellido1
```

FROM depto, profesor
WHERE codigo=cod_dep and depto.nombre='Estadística y Matemática Aplicada'

2) Para insertar registros.

SINTAXIS:

INSERT INTO nombre_de_tabla VALUES (value1, value2,...)

EJEMPLO:

INSERT INTO depto VALUES ('CS', 'Computer Science')

3) Para eliminar registros.

SINTAXIS:

DELETE FROM nombre_de_tabla WHERE condición

EJEMPLO:

DELETE FROM depto WHERE codigo='CS'

4) Para actualizar registros (pueden ser varios).

SINTAXIS:

UPDATE nombre_de_tabla SET cambios WHERE condición

EJEMPLO:

UPDATE depto SET codigo='SE', nombre='Ingenieria de Software'

WHERE codigo='CS'

8.9. Java y BD

Utilizaremos el código que se encarga de leer de una base de datos ya creada.

Necesitaremos importar el paquete `java.sql`.

Lo primero que se hace es hacer una llamada al driver JDBC-ODBC para cargarlo.

Luego se realiza la conexión a nuestro origen de datos. Para tal fin se crea un objeto de tipo `Connection`.

Después se crea la declaración por medio de un objeto de tipo `Statement`.

Se ejecuta la declaración y el resultado se almacena en un objeto de tipo `ResultSet`.

Finalmente se muestran en pantalla los datos de los registros obtenidos como resultado.

EJEMPLO:

Observar el siguiente código. La BD ya debe estar creada y poblada con datos.

```
import java.sql.*;  
  
public class Ejemplo44fp {  
  
    public static void main(String[] args) {  
        try {  
            //cargamos el driver  
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");  
  
            //define la base de datos con la que trabajaremos  
            String sourceURL = new String("jdbc:odbc:technical_library");  
  
            //Crea un objeto connection a través de la clase DriverManager  
            Connection miConexion = DriverManager.getConnection(sourceURL);  
  
            Statement miStatement = miConexion.createStatement();  
            ResultSet losDatos = miStatement.executeQuery("SELECT  
nombre, apellido FROM Persona");  
  
            //Mostramos los datos de los registros en la pantalla  
            while (losDatos.next()) {  
                System.out.println(losDatos.getString("nombre") +  
" " +  
losDatos.getString("apellido"));  
            }  
        } catch (ClassNotFoundException cnfe) {  
            System.err.println(cnfe);  
        } catch (SQLException sqle) {  
            System.err.println(sqle);  
        }  
    }  
}
```

EJERCICIOS PROPUESTOS

1. Investigar sobre la creación de una BD desde Java (no desde el SGBD). Crear una BD de al menos 2 tablas relacionadas sobre cualquier minimundo.
2. Poblar la BD con datos desde Java.
3. Escribir un ejemplo de uso y realizar 2 consultas con la BD creada y poblada.

ANEXO - VIDEOJUEGO DE ESTRATEGIA

Los estudiantes se sienten muy motivados con los videojuegos. Así crearemos un módulo de un videojuego de estrategia.

El videojuego está enmarcado en la época de la caída del Imperio Romano. Un imperio en decadencia que tiene que afrontar la invasión de pueblos bárbaros. El módulo a desarrollar tiene que ver con la parte de las batallas a librar, lo que se llama el aspecto táctico de un juego de estrategia.

El aspecto estratégico queda para futuras experiencias y proyectos más avanzados. Las batallas se libran entre 2 ejércitos. Cada ejército está compuesto por soldados. Según la experiencia en videojuegos, por parte de los estudiantes, se elaboran los siguientes requerimientos:

Para cada soldado nos interesa su nombre, nivel de ataque, nivel de defensa, nivel de vida máxima, su vida actual, velocidad, actitud (defensiva, ofensiva, fuga) y si aún está vivo. También se ubican aquellas acciones que cada soldado podrá realizar, tales como atacar, defender, avanzar, retroceder, ser atacado, huir y morir.

Lo primero a desarrollar es la clase Soldado.

Una vez que tengamos la clase Soldado, podemos crear un ejército de ellos aplicando el mecanismo llamado "composición" de clases. Se creará una nueva clase Ejército que tendrá como parte a un conjunto de Soldados.

Se crea la clase Ejército respectiva en Java (initialmente con todos los soldados con nombres autogenerados, pero con los otros atributos iguales).

Los ejércitos no tendrán un solo tipo de soldado, en cambio ahora podrán tener diferentes tipos de soldados y disponemos de 3 subtipos de la clase Soldado: Espadachín, Caballero y Arquero. Se aplica el concepto de "herencia" de clases, ya que los 3 son subtipos específicos del soldado genérico. Los 3 tienen los mismos atributos y métodos, pero cada uno añade sus particularidades.

Se crea el atributo de clase num para Soldado que va a contar la cantidad de objetos Soldado que se van creando y que aumenta en el constructor de dicha clase, se crea también el método de clase cuantos que devuelve dicho dato.

Para el testing se hace la simulación del juego de estrategia, las culturas se escogen aleatoriamente entre Romanos, Francos, Sajones, Vándalos y Visigodos. La cantidad y subtipo de soldados de cada ejército también se genera aleatoriamente, cantidad entre 1 y 10, y subtipo entre los 3 ya indicados. Se determina al ganador de la batalla en función de la sumatoria de los niveles de vida de todos los soldados de cada ejército, considerando

que según sea el subtipo de soldado tendrá un nivel de vida diferente, Espadachín 10, Caballero 12 y Arquero 7. Se les plantea a los estudiantes, como práctica, que propongan una métrica más realista para decidir al ganador de la batalla. Por ejemplo, podemos ver 2 ejércitos generados: el primero de los Francos con 5 unidades constituidas por 1 arquero, 1 caballero y 3 espadachines. El segundo ejército es de los Romanos formado por 7 unidades: 2 arqueros, 2 caballeros y 3 espadachines.

El objetivo de esta práctica es determinar las clases básicas junto con sus atributos y métodos. Crear su diagrama de clases UML y la aplicación en Java correspondiente.

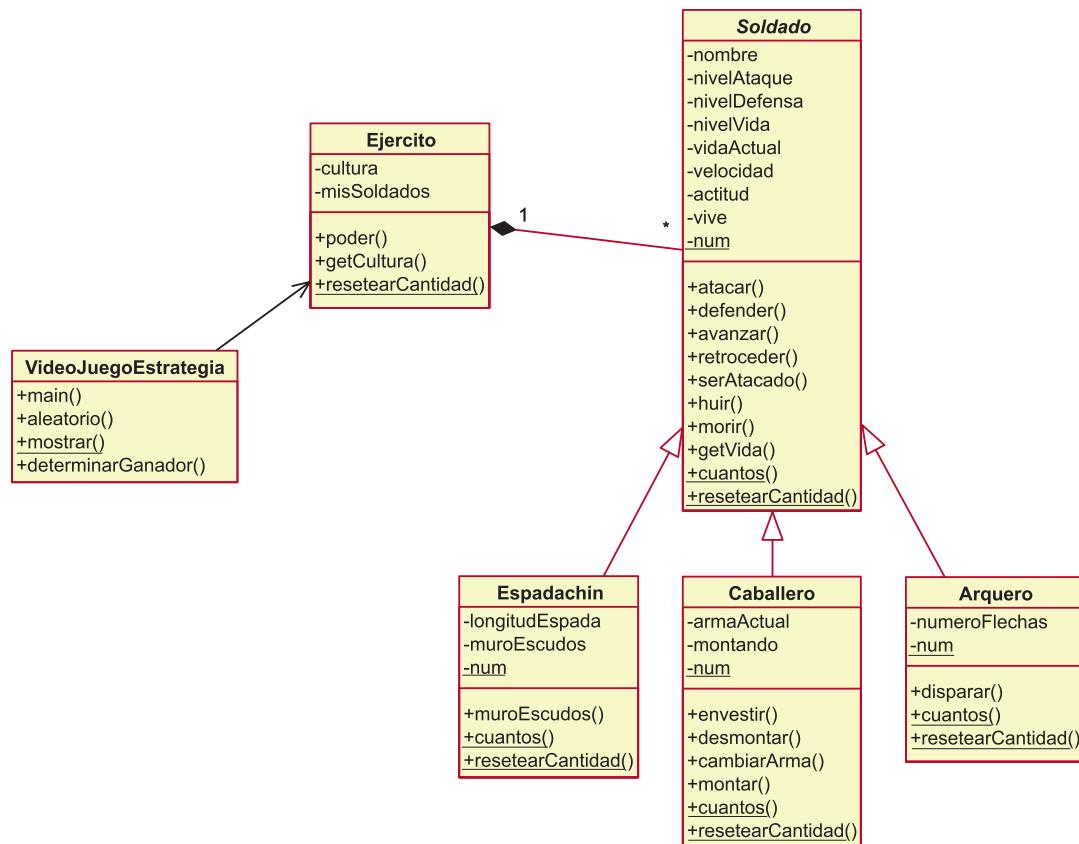


Figura 77. Diagrama de Clases UML del videojuego de estrategia

```

public abstract class Soldado {
    private String nombre;
    private int nivelAtaque;
    private int nivelDefensa;
    private int nivelVida;
    private int vidaActual;
    private int velocidad = 0;
    private String actitud = "defensa";
    private boolean vive = true;
    private static int num = 0;
    public Soldado(String nomb, int ataque, int defensa, int vida) {
        nombre = nomb;
        nivelAtaque = ataque;
        nivelDefensa = defensa;
        nivelVida = vida;
        vidaActual = vida;
        num++;
    }
}
  
```

```

        }
    public void atacar() {
        actitud = "ataque";
        avanzar();
    }
    public void defender() {
        actitud = "defensa";
        velocidad = 0;
    }
    public void avanzar() {
        velocidad++;
    }
    public void retroceder() {
        velocidad--;
    }
    public void serAtacado() {
        vidaActual--;
        if (vidaActual==0)
            morir();
    }
    public void huir() {
        actitud = "fuga";
        velocidad++;
    }
    public void morir() {
        vive = false;
    }
    public String toString(){
        return nombre+ " "+nivelAtaque+ " "+nivelDefensa+ " "+nivelVida+
               "+vidaActual+ " "+velocidad+ " "+ actitud+ " "+vive;
    }
    public static int cuantos() {
        return num;
    }
    public static void resetearCantidad() {
        num=0;
    }
    public int getVida(){
        return nivelVida;
    }
}
public class Arquero extends Soldado{

    private int numeroFlechas;
    private static int num = 0;

    public Arquero(String s, int ata, int def, int vid, int cant) {
        super(s,ata,def,vid);
        numeroFlechas = cant;
        num++;
    }
    public void disparar(){
        if (numeroFlechas>0)
            numeroFlechas--;
    }
    public static int cuantos() {
        return num;
    }
}

```

```

        }
    public static void resetearCantidad() {
        num=0;
    }
    public String toString(){
        return super.toString()+" "+numeroFlechas;
    }
}
public class Caballero extends Soldado{

    private String armaActual = "Lanza";
    private boolean montando = true;
    private static int num = 0;

    public Caballero(String s, int ata, int def, int vid){
        super(s,ata,def,vid);
        num++;
    }
    public void envestir(){
        if (montando==true)
            for(int i=0;i<=2;i++)
                super.atacar();
        else
            super.atacar();
    }

    public void desmontar(){
        if(montando==true){
            montando = false;
            super.defender();
            cambiaArma();
        }
    }
    public void cambiaArma(){
        if(armaActual=="Lanza")
            armaActual = "Espada";
        else
            armaActual = "Lanza";
    }
    public void montar(){
        if(montando==false){
            montando = true;
            super.atacar();
            cambiaArma();
        }
    }
    public static int cuantos(){
        return num;
    }
    public static void resetearCantidad(){
        num=0;
    }
    public String toString(){
        return super.toString()+" "+armaActual+ " "+montando;
    }
}
public class Espadachin extends Soldado{

```

```

private int longitudEspada;
private boolean muroEscudos = false;
private static int num = 0;

public Espadachin(String s, int ata, int def, int vid, int lon) {
    super(s,ata,def,vid);
    longitudEspada = lon;
    num++;
}
public void muroEscudos() {
    if(muroEscudos==true)
        muroEscudos = false;
    else
        muroEscudos = true;
}
public static int cuantos() {
    return num;
}

public static void resetearCantidad() {
    num=0;
}
public String toString(){
    return super.toString()+" "+longitudEspada+ " "+muroEscudos;
}
import java.util.*;

public class Ejercito {

    ArrayList<Soldado> misSoldados = new ArrayList<Soldado>();
    String cultura;

    public Ejercito(String cult, int cantidad) {
        cultura = cult;
        int tipo;
        for(int i=0;i<cantidad;i++){
            tipo = (int) (Math.random()*3)+1;
            switch (tipo) {
                case 1: misSoldados.add(new Espadachin("E"+i,10,8,10,40));
                break;
                case 2: misSoldados.add(new Caballero("C"+i,13,7,12));
                break;
                case 3: misSoldados.add(new Arquero("A"+i,7,3,7,20));
                break;
            }
        }
    }
    public String toString(){
        String todos = "";
        for(int i=0;i<misSoldados.size();i++)
            todos += misSoldados.get(i)+" ";
        return cultura+" "+misSoldados.size()+" "+todos;
    }
    public int poder(){
        int poder = 0;
        for(int i=0;i<misSoldados.size();i++)
    }
}

```

```

        poder += misSoldados.get(i).getVida();
        return poder;
    }
    public String getCultura(){
        return cultura;
    }
    public static void resetearCantidad(){
        Soldado.resetearCantidad();
        Arquero.resetearCantidad();
        Caballero.resetearCantidad();
        Espadachin.resetearCantidad();
    }
}
public class VideoJuegoEstrategia {

    public static void main(String[] args) {
        int cant;
        String
        cultura[]={"Romanos","Francos","Sajones","Visigodos","Vandalos"};
        cant = aleatorio(1,10);
        Ejercito e1 = new Ejercito(cultura[aleatorio(0,4)],cant);
        mostrar(e1);
        cant = aleatorio(1,10);
        Ejercito e2 = new Ejercito(cultura[aleatorio(0,4)],cant);
        mostrar(e2);
        determinarGanador(e1,e2);
    }
    public static int aleatorio(int a,int b){
        return (int) (Math.random() * (b-a+1))+a;
    }
    public static void mostrar(Ejercito e){
        System.out.print(e);
        System.out.println("Cantidad total de soldados:"+Soldado.
cuantos()+"\n"+
"Espadachines: "+Espadachin.cuantos()+"\n"+Arqueros:
" +
Arquero.cuantos()+"\n"+Caballeros: "+Caballero.
cuantos());
        Ejercito.resetearCantidad();
    }
    public static void determinarGanador(Ejercito e1, Ejercito e2){
        System.out.println("Ejercito 1: "+e1.getCultura()+" "+e1.
poder());
        System.out.println("Ejercito 2: "+e2.getCultura()+" "+e2.
poder());
        if (e1.poder()>e2.poder())
            System.out.println("El ganador es ejercito1 de : "+e1.
getCultura());
        else if (e1.poder()<e2.poder())
            System.out.println("El ganador es ejercito2 de : "+e2.
getCultura());
        else
            System.out.println("Sin ganador!!!");
    }
}
}

```

BIBLIOGRAFÍA

- Aedo, M., Vidal, E., Castro E. & Paz, A. (2017). "Aproximación orientada a Entornos Lúdicos para la primera sesión de CS1 - Una experiencia con nativos digitales". *Proceedings of 15th Latin American and Caribbean Conference for Engineering and Technology (LACCEI) - International Multi-Conference for Engineering, Education, and Technology*. United States.
- Aedo, M. (2019). *Fundamentos de Programación 1 – Java Básico*. Editorial UNSA.
- Aedo, M., Vidal, E. & Castro, E. (2019). "Experiencia en la enseñanza de Fundamentos de Programación Orientada a Objetos a través de la implementación de un Videojuego de Estrategia". *Proceedings of 17th Latin American and Caribbean Conference for Engineering and Technology (LACCEI) - International Multi-Conference for Engineering, Education, and Technology*. Jamaica.
- Bloch, J. (2018). *Effective Java* (3rd ed). Addison-Wesley Professional.
- Dean, J. (2008). *Introduction to Programming with Java*. McGraw-Hill.
- Deitel, P. (2014). *Java SE for Programmers* (3rd ed.). Prentice Hall.
- Deitel, P. (2017). *Java How to Program. Early Objects* (11th ed.). Prentice Hall.
- Deitel, P. (2017). *Java How to Program. Late Objects* (11th ed.). Prentice Hall.
- Wu, T. (2010). *An Introduction to Object-Oriented Programming with Java* (5th ed.). McGraw-Hill.

