Ingeniería de sistemas

PROGRAMACIÓN ORIENTADA A OBJETOS EN JAVA

Buenas prácticas

Miguel Hernández Bejarano
Luis Eduardo Baquero Rey

eduU

Ingeniería de sistemas

PROGRAMACIÓN ORIENTADA A OBJETOS EN JAVA

Buenas prácticas

Miguel Hernández Bejarano
Luis Eduardo Baquero Rey

ediU

Informática

PROGRAMACIÓN ORIENTADA A OBJETOS EN JAVA

Buenas prácticas

Miguel Hernández Bejarano
Luis Eduardo Baquero Rey

Conocimiento a su alcance
Bogotá - México, D.F.

Hernández Bejarano, Miguel, *et al.*

Programación orientada a objetos en Java/ Miguel Hernández Bejarano y Luis Eduardo Baquero Rey -- 1a. Edición. Bogotá : Ediciones de la U, 2023.
316 p. ; 24 cm.
ISBN 978-958-792-463-3 e-ISBN 978-958-792-464-0
1. Informática 2. Lenguajes de programación 3. Programación orientada a objetos en Java I. Tít.
621.39 ed.

Área: Informática
Primera edición: Bogotá, Colombia, enero de 2023
ISBN. 978-958-792-463-3

© Miguel Hernández Bejarano
© Luis Eduardo Baquero Rey
© Ediciones de la U - Carrera 27 # 27-43 - Tel. (+57-1) 3203510 - 3203499
www.edicionesdelau.com - E-mail: editor@edicionesdelau.com
Bogotá, Colombia

Ediciones de la U es una empresa editorial que, con una visión moderna y estratégica de las tecnologías, desarrolla, promueve, distribuye y comercializa contenidos, herramientas de formación, libros técnicos y profesionales, e-books, e-learning o aprendizaje en línea, realizados por autores con amplia experiencia en las diferentes áreas profesionales e investigativas, para brindar a nuestros usuarios soluciones útiles y prácticas que contribuyan al dominio de sus campos de trabajo y a su mejor desempeño en un mundo global, cambiante y cada vez más competitivo.

Coordinación editorial: Adriana Gutiérrez M.
Carátula: Ediciones de la U
Impresión: DGP Editores SAS
Calle 63 No. 70 D - 34, Pbx. (571) 7217756

Impreso y hecho en Colombia
Printed and made in Colombia

No está permitida la reproducción total o parcial de este libro, ni su tratamiento informático, ni la transmisión de ninguna forma o por cualquier medio, ya sea electrónico, mecánico, por fotocopia, por registro y otros medios, sin el permiso previo y por escrito de los titulares del Copyright.

Dedicatoria

A Dios, a mi esposa Flor Ángela,
a mis hijos Jefferson, Julieth y Óscar
y a mi nieto Alejandro
Miguel

A Cristal, Isabela y Rocío
Eduardo

Contenido

Prólogo 17

Capítulo 1. Fundamentos de la Programación Orientada a Objetos (POO) 21

1.1 Temática a desarrollar 21
1.2 Introducción 21
1.3 Programación orientada a objetos 23
1.4 Principios de la programación orientada a objetos 23
1.5 Ventajas del uso de la programación orientada a objetos 24
1.6 Desventajas del uso de la programación orientada a objetos 24
1.7 Lenguajes de programación orientada a objetos 24
1.8 Clases y objetos 25
    1.8.1 Clases 25
    1.8.2 Objeto 29
1.9 Visibilidad en atributos y métodos 34
1.10 Paquetes 35
1.11 Lecturas recomendadas 36
1.12 Preguntas revisión de conceptos 37
1.13 Ejercicios 37
1.14 Bibliografía 38

Capítulo 2. Componentes de un programa 39

2.1 Temática a desarrollar 39
2.2 Introducción 39
2.3 Características de un programa en Java 40
2.4 Estructura de un programa en Java 40
    2.4.1 Declaración de importaciones (import) 40
    2.4.2 Definición de clases 41

2.5 Presentación del programa 43
2.6 Construcción de un programa en Java 43
2.7 Palabras reservadas 44
2.8 Los identificadores 45
2.9 Tipos de datos 45
2.10 Variables 47
2.11 Constantes 48
2.12 Comentarios 48
2.13 Definición de variables 50
2.14 Definición de constantes 50
2.15 Operadores aritméticos 51
2.16 Los separadores 52
2.17 Lecturas recomendadas 52
2.18 Preguntas de revisión de conceptos 52
2.19 Ejercicios 53
2.20 Referencias bibliográficas 53

Capítulo 3. Herramientas para el desarrollo de la programación orientada a objetos 55

3.1 Temática a desarrollar 55
3.2 Introducción 55
3.3 Entornos Integrados de Desarrollo (IDE) 56
3.3.1 Eclipse 57
3.3.2 NetBeans 57
3.3.3 BlueJ 58
3.3.4 GreenFoot 59
3.3.5 IntelliJ IDEA 59
3.4 Herramientas de modelamiento 61
3.4.1 StarUML 61
3.4.2 Dia 61
3.4.3 ArgoUML 61
3.4.4 Rational Software Modeler 62
3.5 Java como lenguaje de programación 62
3.6 Lecturas recomendadas 64
3.7 Preguntas de revisión de conceptos 64
3.8 Ejercicios 65

3.9 Referencias bibliográficas 65

Capítulo 4. Métodos Métodos constructores, set y get 67

4.1 Temática a desarrollar 67
4.2 Introducción 67
4.3 Métodos 68
4.3.1 Método constructor 68
4.4 Modificadores de acceso 72
4.5 Métodos set y get 78
4.6 Métodos de instancias 83
4.7 Estructura de un método 84
4.8 Prueba de escritorio 85
4.9 Métodos que no retornan valor 88
4.10 Requerimientos 88
4.11 Un desarrollo completo 89
4.12 Entrada y salida de datos 95
4.13 Entrada y salida estándar 96
4.14 Lecturas recomendadas 106
4.15 Preguntas de revisión de conceptos 106
4.16 Ejercicios 107
4.17 Bibliografía 107

Capítulo 5. Condicionales 109

5.1 Temática a desarrollar 109
5.2 Introducción 109
5.3 Estructuras de control condicionales 110
5.4 Operadores relacionales 110
5.5 Operadores lógicos 112
5.6 Tipos de condicionales 114
5.6.1 Condicionales simples 114
5.6.2 Condicionales compuestas 119
5.6.3 Condicionales anidadas 122
5.7 Selección múltiple 131
5.8 Operador condicional (?) 135
5.9 Preguntas de revisión de conceptos 138

5.10 Lecturas recomendadas 139
5.11 Ejercicios 139
5.12 Bibliografía 142

Capítulo 6. Ciclos 143

6.1 Temática a desarrollar 143
6.2 Introducción 143
6.3 Estructuras de control repetitivas o ciclos 145
6.3.1 Estructura de repetición while (mientras) 145
6.3.2 Estructura de repetición do-while 152
6.3.3 Estructura de repetición for 156
6.4 Comparación de los ciclos while, do-while, for 160
6.5 Ciclo for-each (for mejorado) 161
6.6 Uso de los ciclos repetitivos 162
6.7 Lecturas recomendadas 162
6.8 Preguntas y ejercicios de revisión de conceptos 163
6.9 Ejercicios 163
6.10 Bibliografía 163

Capítulo 7. Cadenas 165

7.1 Temática a desarrollar 165
7.2 Introducción 165
7.3 Clase String 167
7.4 Obtener cadenas desde las primitivas 174
7.5 Obtener primitivas desde las cadenas 174
7.6 Lecturas recomendadas 184
7.7 Preguntas de revisión de conceptos 184
7.8 Ejercicios 185
7.9 Referencias bibliográficas 186

Capítulo 8. Relaciones entre clases 189

8.1 Temática a desarrollar 189
8.2 Introducción 189
8.3 Elementos entre las relaciones de clases 191

8.3.1 Cardinalidad 191
8.3.2 Asociación 192
8.3.3 Agregación 194
8.3.4 La composición 196
8.4 La relación de especialización/generalización 197
8.5 Dependencia 198
8.6 Lecturas recomendadas 199
8.7 Preguntas de evaluación 199
8.8 Ejercicios 199
8.9 Bibliografía 202

Capítulo 9. Abstracción y encapsulamiento 203

9.1 Temática a desarrollar 203
9.2 Introducción 203
9.3 Abstracción 203
9.4 Encapsulamiento 206
9.5 Lecturas recomendadas 209
9.6 Preguntas de revisión de conceptos 209
9.7 Ejercicios 210
9.8 Bibliografía 211

Capítulo 10. Herencia 213

10.1 Temática a desarrollar 213
10.2 Introducción 213
10.3 Herencia 215
10.4 Representación de la herencia 219
10.5 Sentencia super 225
10.6 Lecturas recomendadas 228
10.7 Preguntas de revisión de conceptos 228
10.8 Ejercicios 229
10.9 Referencias bibliográficas 231

Capítulo 11. Polimorfismo 233

11.1 Temática a desarrollar 233

11.2 Introducción 233
11.3 Polimorfismo 234
11.4 Sobrecarga de métodos 234
11.5 Sobrecarga de constructores 238
11.6 Sobreescritura de métodos 241
11.7 Enlace dinámico 247
11.8 Lecturas recomendadas 250
11.9 Preguntas de revisión de conceptos 251
11.10 Ejercicios 251
11.11 Bibliografía 252

Capítulo 12. Interfaz y clases abstractas 253

12.1 Temática a desarrollar 253
12.2 Introducción 253
12.3 Interfaz 254
12.4 Clases abstractas 258
12.5 Preguntas de revisión de conceptos 260
12.6 Ejercicios 261
12.7 Bibliografía 265

Capítulo 13. Modelado orientado a objetos y aplicaciones de software 267

13.1 Temática a desarrollar 267
13.2 Introducción 267
13.3 Análisis orientado a objetos 268
13.4 Escenarios 269
13.5 Prototipo 270
13.6 Diagramas de casos de uso 270
13.7 Diseño orientado a objetos 277
13.8 Diagramas UML (Unified Modeling Language) 280
13.9 Aplicaciones de software 281
13.10 Lecturas recomendadas 299
13.11 Preguntas de revisión de conceptos 299
13.12 Ejercicios 299
13.13 Bibliografía 301

Capítulo 14. Documentación 303

14.1 Temática a desarrollar..... 303
14.2 Introducción 303
14.3 Javadoc 303
14.4 Javadoc en los IDE 309
14.5 API de Java 313
14.6 Lecturas recomendadas 313
14.7 Preguntas de revisión de conceptos..... 313
14.8 Ejercicios 314
14.9 Bibliografía 315

Índice de figuras

Figura 1. Mapa mental de programación orientada a objetos 22
Figura 2. Mapa mental de clase 26
Figura 3. Partes del diagrama de clase 27
Figura 4. Mapa mental de objeto 30
Figura 5. Ejemplos del objeto reloj 32
Figura 6. Diagrama de paquete 36
Figura 7. Diagrama de paquetes con dos clases comunes 36
Figura 8. Mapa mental de tipos de datos 46
Figura 9. Pantalla de carga de Eclipse LUNA 57
Figura 10. Pantalla inicio en NetBeans 58
Figura 11. Pantalla de inicio de BlueJ 58
Figura 12. Pantalla inicio de GreenFoot 59
Figura 13. Pantalla IntelliJ IDEA 60
Figura 14. Concepto de Java 63
Figura 15. Concepto plataformas de Java 64
Figura 16. Mapa mental del concepto de métodos 69
Figura 17. Mapa mental de método constructor 70
Figura 18. Mapa mental de los métodos set y get 79
Figura 19. Concepto de las estructuras condicionales 111
Figura 21. Mapa mental de ciclos 144
Figura 20. Mapa mental de la clase String 166
Figura 22. Mapa mental de relaciones entre clases 190
Figura 23. Mapa mental de abstracción 205
Figura 24. Mapa mental de encapsulamiento 207
Figura 25. Mapa mental de herencia 214
Figura 26. Mapa mental de polimorfismo 235
Figura 27. Mapa mental del modelado orientado a objetos 268
Figura 27a. Mapa mental de aplicaciones de software 283
Figura 28. Documentación en BlueJ 309
Figura 29. Generación Javadoc en Eclipse 310
Figura 30. Documentación Javadoc en Eclipse 311
Figura 31. Generación Javadoc en Netbeans 312
Figura 32. Documentación Javadoc en Netbeans 312
Figura 33. API Java 8 314

Índice de tablas

Tabla 1. Ejemplos de objetos asociados a su correspondiente clase..... 25
Tabla 2. Ejemplo de objeto 29
Tabla 3. Ejemplo estructura interna de un objeto 34
Tabla 4. Palabras reservadas en Java 44
Tabla 5. Tipos de datos de Java..... 47
Tabla 6. Operadores aritméticos 51
Tabla 7. Operadores relacionales..... 112
Tabla 8. Operadores lógicos 113
Tabla 9. Resultado operadores lógicos 113
Tabla 10. Apoyo implementación del método 117
Tabla 11. Apoyo parte de análisis 126
Tabla 12. Ejemplo de cadenas..... 167
Tabla 13. Métodos básicos 168
Tabla 14. Tipos de cardinalidad 191
Tabla 15. Comparación entre abstracción y encapsulamiento..... 209

Prólogo

En el quehacer diario de la docencia, y más concretamente en el área de la programación, surgen muchos temas, talleres y ejercicios para trabajar con los estudiantes. En tal sentido, este libro es el resultado de varios años de experiencias donde se retroalimentan carencias y éxitos en busca de buenas prácticas de la Programación Orientada a Objetos (POO); para tal efecto, se utiliza Java como lenguaje de programación de alto nivel. Está diseñado para cualquier estudiante que desea hacer inmersión en la programación de computadores, independientemente del área o disciplina del conocimiento, pero, eso sí, cambiando el paradigma de la programación estructurada a la de los objetos, como en la vida real.

Se tienen en cuenta elementos de buenas prácticas de la programación con la finalidad de hacer fácil la lectura del código, así como su mantenimiento, facilitando la escalabilidad del mismo, la reutilización y la integración de manera homogénea, estandarizando el desarrollo, basado en convenios y reglas sencillas, y aportando higiene en la codificación de los programas fuente.

Con el propósito de lograr lo anterior, esta obra se divide en catorce capítulos, cada uno distribuido de la siguiente manera: relación general de la temática a desarrollar, una introducción al tema central donde se complementa con un mapa mental, el desarrollo de la temática respectiva incluyendo ejemplos, se recomiendan unas lecturas que posibilitan ampliar el tema, unas preguntas de revisión de conceptos, ejercicios propuestos y, finalmente, se propone una bibliografía complementaria.

En el capítulo 1, se tratan temas fundamentales de este paradigma, como el de la programación orientada a objetos, los principios de la POO y los conceptos de clase, objeto, atributo y método.

En el capítulo 2, se estudian los componentes de un programa, donde se involucran identificadores, tipos de datos, palabras reservadas, manejo de comentarios, variables, constantes, las expresiones, la estructura de un programa de Java, la entrada y la salida de datos y los tipos de operadores.

En el capítulo 3, se estudian las herramientas necesarias para el desarrollo de la programación orientada a objetos, como los entornos integrados de desarrollo, herramientas de modelamiento y el lenguaje de programación Java.

En el capítulo 4, se hace referencia a los métodos, qué son los métodos constructores, los métodos set y get y los métodos modificadores.

En el capítulo 5, se estudia todo lo relacionado con los condicionales, en su orden, las estructuras de control, los condicionales simple, compuesto y anidado, los operadores lógicos, selección múltiple y el operador condicional ?.

En el capítulo 6, se estudia el tema relacionado con el manejo de cadenas en Java, incluyendo los métodos asociados a la clase String.

En el capítulo 7, se hace referencia a los ciclos o instrucciones repetitivas que se pueden trabajar en un programa Java.

En el capítulo 8, se estudian las diferentes relaciones entre clases, como la cardinalidad, asociación, agregación, composición, generalización y dependencia.

En el capítulo 9, se involucran los temas de abstracción y encapsulamiento, como mecanismos de la programación orientada a objetos.

En el capítulo 10, se estudia el concepto de herencia y la terminología asociada a este concepto, como lo que es una superclase, subclase, protected, extends y super.

En el capítulo 11, se hace referencia al mecanismo de la programación orientada a objetos denominado polimorfismo, incluyendo polimorfismo de subtipado, sobrecarga de métodos y sobreescritura de métodos.

En el capítulo 12, se continúa con el concepto y manejo de interfaz y clases abstractas en Java.

En el capítulo 13, se estudian los aspectos básicos a tener en cuenta a la hora de desarrollar un proyecto o aplicación de software de comienzo a fin, desde el análisis orientado a objetos, el diseño, el modelamiento y la aplicación de software final.

En el capítulo 14, por último, se presentan opciones desde el lenguaje de programación Java para la respectiva documentación de los programas y

aplicaciones, con herramientas con Javadoc, soporte Javadoc para los IDE y la API de Java.

El estudiante interesado en el tema podría orientarlo también a cualquier otro lenguaje de programación de alto nivel, diseñado para trabajar con el paradigma de la programación orientada a objetos.

Capítulo 1

Fundamentos de la Programación Orientada a Objetos (POO)

1.1 Temática a desarrollar

- Programación orientada a objetos
- Principios de la POO
- Clase
- Objeto
- Atributo
- Método

1.2 Introducción

La concepción del mundo en términos de Programación Orientada a Objetos (POO) está integrada por un conjunto de entidades u objetos relacionados entre sí, los cuales pueden ser: personas, árboles, automóviles, casas, teléfonos y computadores; en general, todo elemento que tiene atributos y que pueden ser palpables a través de los sentidos.

El ser humano tiene una apreciación de su ambiente en términos de objetos, por lo cual le resulta simple pensar de la misma manera cuando diseña un modelo para la solución de un problema, puesto que se tiene el concepto o pensamiento de objetos de manera intrínseca.

La Programación Orientada a Objetos (*Object Oriented Programming*) tiene como ejes centrales las clases y los objetos, donde una clase corresponde a la agrupación de datos (atributos) y acciones (métodos). Una clase describe un conjunto de objetos con propiedades y comportamientos similares. La figura 1 muestra, mediante un concepto de mapa mental, este concepto y todo lo que le rodea.

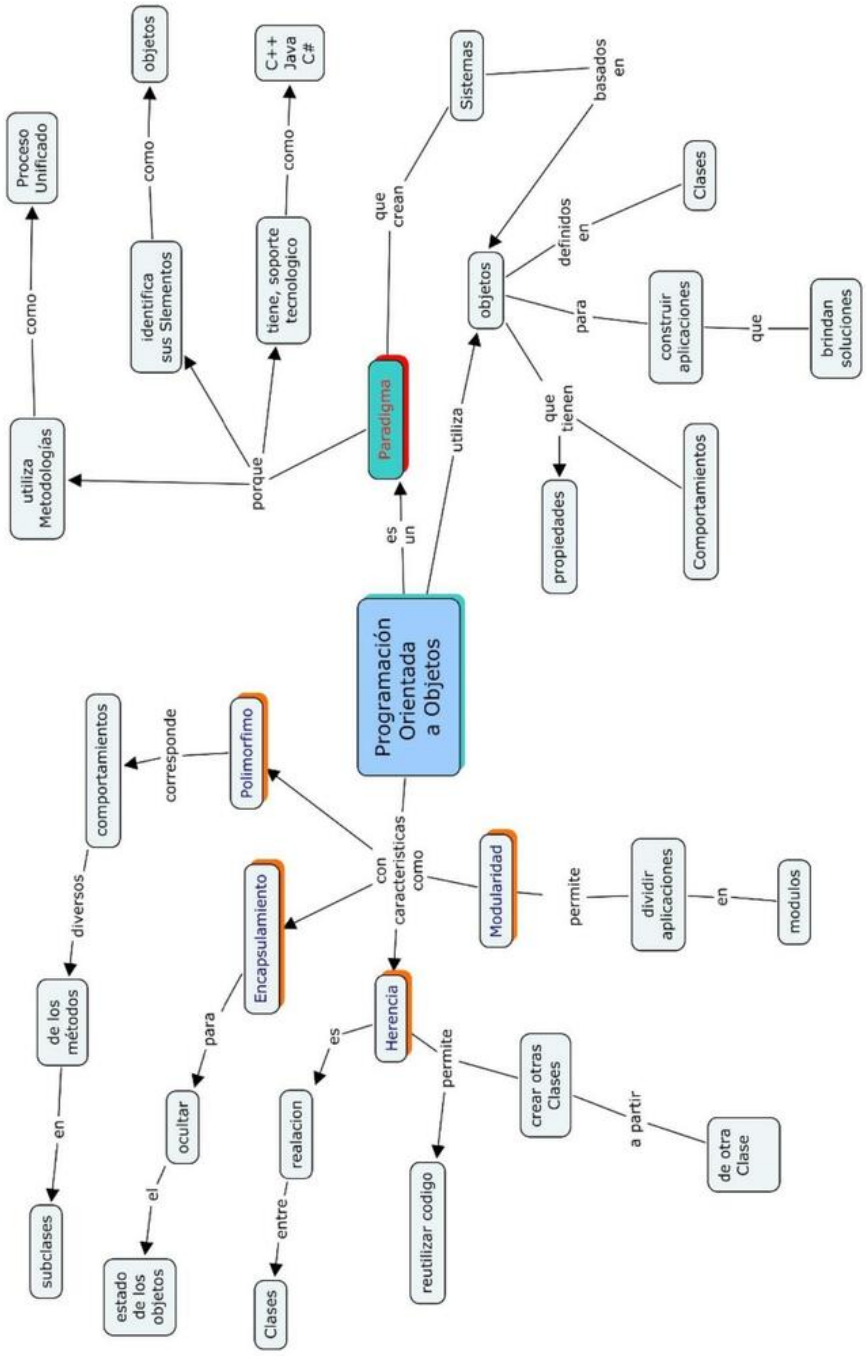


Figura 1
Mapa mental de programación orientada a objetos

El buen uso de este paradigma de programación se convierte en una buena práctica que arrojaría como resultados la construcción de programas de calidad, facilitar su mantenimiento y la reutilización de programas, entre otros aspectos. El tomar objetos de un problema del mundo real y extraer sus características y comportamientos, y representarlos formalmente en diagramas UML, mediante los diagramas de clases y diagramas de objetos, son de las primeras actividades inmersas en este estilo de programación.

1.3 Programación orientada a objetos

La Programación Orientada a Objetos (POO) es un paradigma *o modelo de programación;* esto indica que no es un lenguaje de programación específico o una tecnología, sino una *forma de programar* donde lo que se intenta es llevar o modelar el mundo real a un conjunto de entidades u objetos que están relacionados y se comunican entre ellos como elementos constitutivos del software, dando solución a un problema en términos de programación.

En síntesis, el elemento básico de este paradigma es el objeto, el cual contiene toda la información necesaria que se abstrae del mundo del problema, los datos que describen su estado y las operaciones que pueden modificar dicho estado, determinando las capacidades del objeto.

1.4 Principios de la programación orientada a objetos

Abstracción: es la capacidad de determinar los detalles fundamentales de un objeto mientras se ignora el entorno; por ejemplo, la información de un cliente en una factura requiere el documento, el nombre, la dirección y el teléfono.

Encapsulamiento: es la capacidad de agrupar en una sola unidad datos y métodos. Por ejemplo, para calcular el área de un triángulo, se deben tener la base y la altura; o cuando se ve la televisión, no hay preocupación por el modo cómo este funciona, lo único que se realiza es manipular el control y disfrutar del programa, película, entre otros ejemplos.

Herencia: es el proceso en el que un objeto adquiere las propiedades de otro a partir de una clase base, de la cual se heredan todas sus propiedades, métodos y eventos; estos, a su vez, pueden o no ser implementados y/o modificados;

por ejemplo, la herencia de bienes que se transmite de padre a hijo, pero el padre tiene la potestad de desheredar.

Polimorfismo: es una propiedad que permite enviar el mismo mensaje a objetos de diferentes clases de forma que cada uno de ellos responde al mismo mensaje de diferente modo; por ejemplo, el acto de comer para proveer nutrientes al organismo es diferente en los siguientes animales: el león (carnívoro), el canario (alpiste) y el perro (todo tipo de alimento).

Modularidad: es la propiedad que permite dividir una aplicación de software en unidades o partes más pequeñas, denominadas módulos, donde cada una de ellas puede ser independiente de la aplicación y de los restantes módulos. Por ejemplo, el software institucional de una empresa que puede ser dividido en subprogramas como contabilidad, presupuesto, facturación, entre otros.

1.5 Ventajas del uso de la programación orientada a objetos

- Simula el sistema del mundo real.
- Facilita el mantenimiento del software.
- Facilita el trabajo en equipo.
- Modela procesos de la realidad.
- Genera independencia e interoperabilidad de la tecnología.
- Incrementa la reutilización del software.

1.6 Desventajas del uso de la programación orientada a objetos

- Complejidad para adaptarse.
- Mayor cantidad de código (aunque se tiene la posibilidad de volver, de ser reutilizable).

1.7 Lenguajes de programación orientada a objetos

Permite el desarrollo de aplicativos de software que interactúan con los programas como conjuntos de objetos que se ayudan entre ellos para realizar acciones.

Ejemplo

- C++
- ADA
- Java
- SmallTalk
- C#
- Php versión 5.0 en adelante

1.8 Clases y objetos

Las palabras "clase" y "objeto" son utilizadas en la programación orientada a objetos, la cual planea simular u organizar datos en conjuntos modulares de elementos de información del mundo real.

1.8.1 Clases

Asumida como un molde o plantilla empleada para crear objetos, lo que permite considerar que las clases agrupan a un conjunto de objetos similares.

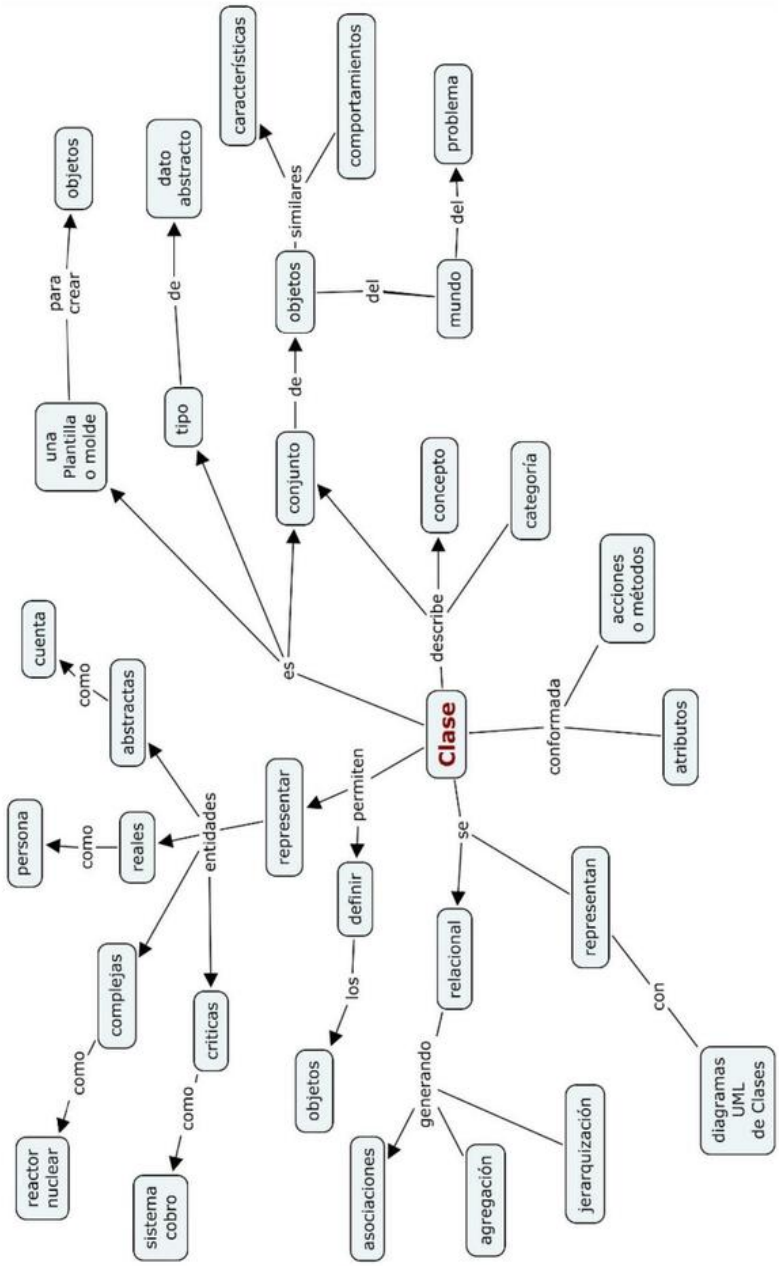
Ejemplo

Tabla 1.
Ejemplos de objetos asociados a su correspondiente clase

Objetos	Clase
Mazda 626 sdk23, rojo Ford, capacidad 32 personas	Auto
• Cra. 15 No. 85 – 19 bloque 8 • Calle 128 # 52 – 90 bloque 12 apto. 402.	Vivienda
39.652.123 Claudia Ramírez	Estudiante
• Garfield (Felino) • Snoopy (Perro) • Simba (Rey León) • Willy (Cetáceo)	Mamífero
• Código 1478 Estructura de Datos • Código 18156 Matemáticas discretas	Asignatura

La figura 2 amplía el concepto de clase.

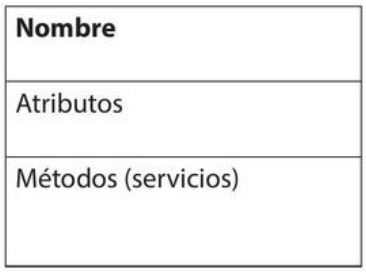
Figura 2.
Mapa mental de clase



1.8.1.1 Representación de las clases

La clase encapsula la información de uno o varios objetos a través de la cual se modela el entorno en estudio. Se representa haciendo uso de los diagramas de clases de UML, conformados por un rectángulo que tiene tres divisiones, como se muestra en la figura 3.

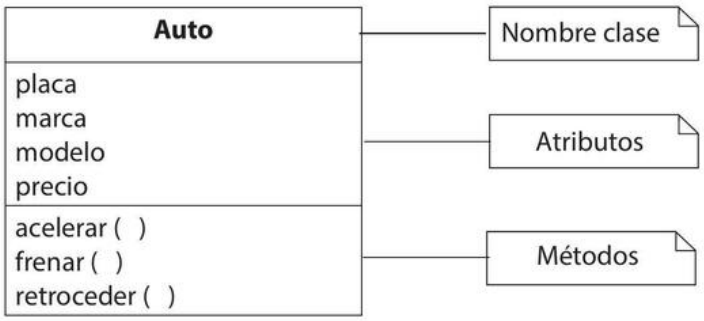
Figura 3.
Partes del diagrama de clase

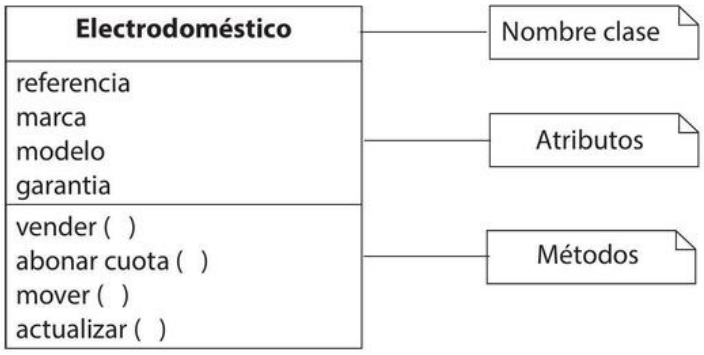


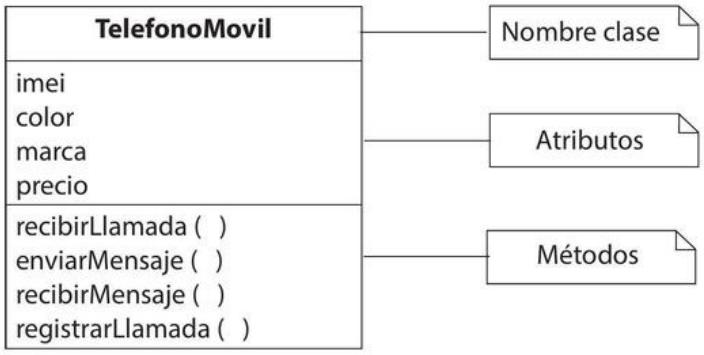
UML (*Unified Modeling Language*) es un lenguaje que permite modelar, construir y documentar los elementos que forman un sistema software orientado a objetos.

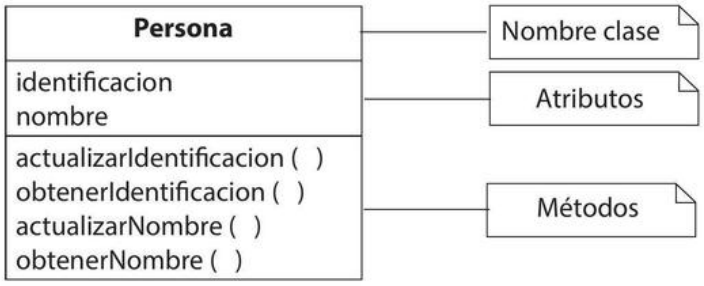
Ejemplo

Los siguientes diagramas de clases asocian la información correspondiente de acuerdo con el número del cuadrante, así, en el primer cuadrante se registra el nombre de la clase, en el segundo cuadrante los nombres de los atributos y en el tercer cuadrante los nombres de los métodos, comportamientos o responsabilidades que realizan.









Los ejemplos anteriores reprentan diagramas de clases con sus correspondientes atributos y métodos; los rectángulos de los lados unidos con líneas punteadas corresponden a la documentación.

Durante del desarrollo de la temática, incluyendo los capítulos siguientes, se irá ampliando la información correspondiente al tema de las clases.

1.8.2 Objeto

Es cualquier cosa real (tangible o intangible) del mundo que nos rodea; por ejemplo, un auto, una casa, un árbol, un reloj, una estrella, una cuenta de ahorro. Un objeto se puede considerar como una entidad compleja provista de propiedades o características y de comportamientos o estados. La figura 4 *(véase en pág. siguiente)* detalla el concepto de objeto de una manera más somera.

A un objeto lo identifican las siguientes características:

Estado

Conformado por el conjunto de propiedades o atributos de un objeto correspondiente a los valores que pueden tomar cada uno de esos atributos.

Ejemplo

El objeto cliente tiene las siguientes características: documento, nombres, apellidos, dirección y teléfono.

Tabla 2.
Ejemplo de objeto

Atributo	Estado
Documento:	10.265.254
Nombres:	Luis Eduardo
Apellidos:	Baquero Rey
Teléfono:	320 256 32 18

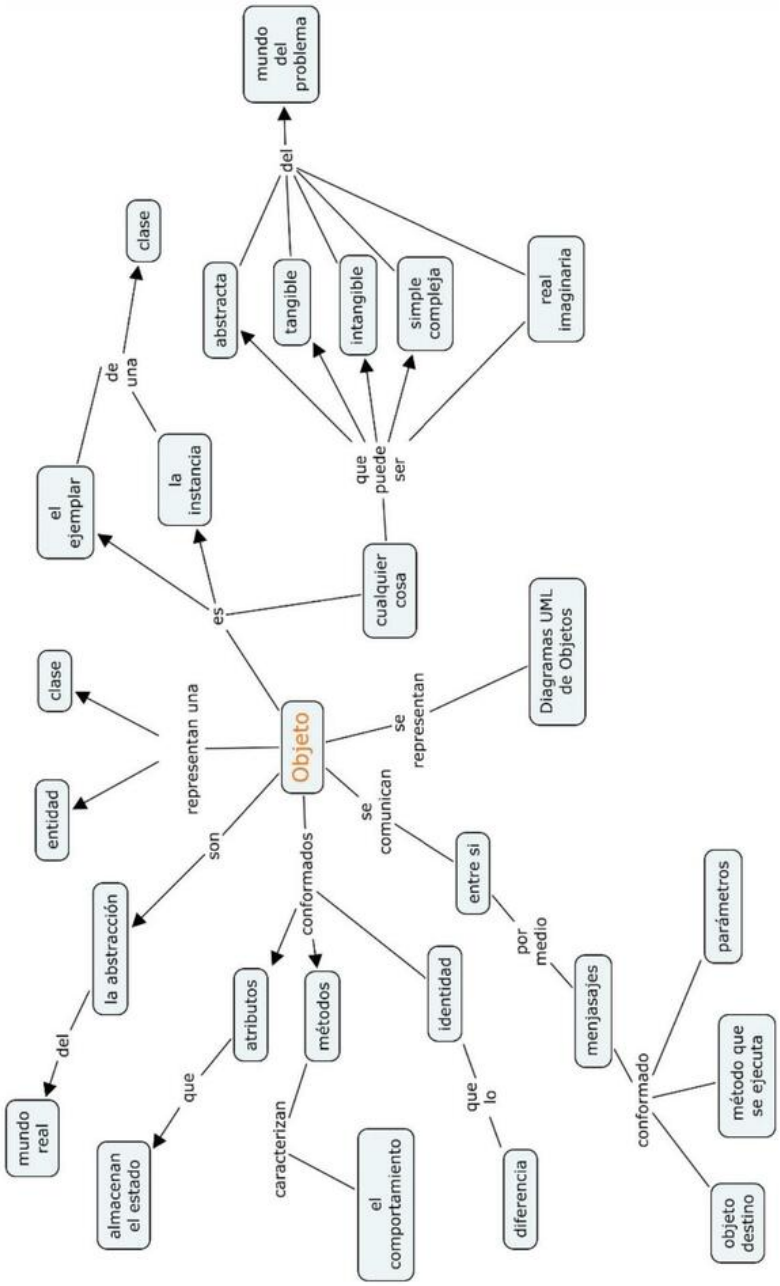


Figura 4.
Mapa mental de objeto

Comportamiento

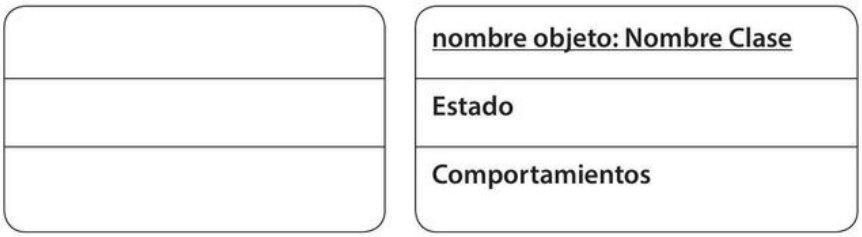
De un objeto, está relacionado con la funcionalidad y determina las operaciones que este puede realizar, responder o ejecutar alguna acción ante mensajes enviados por otros objetos, por ejemplo, actualizar el teléfono de un cliente, realizar la suma de dos números.

Identidad

Corresponde a la propiedad de un objeto que le distingue de todos los demás, como es el caso del ejemplo del cliente Luis Eduardo Baquero Rey, que tiene un número de documento que lo identifica, una dirección y un teléfono.

1.8.2.1 Representación gráfica de los objetos

Los objetos se pueden representar con diagramas UML de objeto, que es un rectángulo con tres divisiones.



Se pueden confundir los términos clase y objeto; en general, una clase es una representación abstracta de algo, mientras que un objeto es una representación de ese algo conformado por la clase.

Se tienen cuatro objetos correspondientes a relojes (de pared, arena, digital y de pulsera). La clase Reloj tiene las características que agrupa a esa colección de objetos.

Figura 5.
Ejemplos del objeto reloj

A continuación, se modela la clase Reloj y un objeto reloj mediante los diagramas correspondientes.

Reloj
hora minuto
Reloj ( ) fijarHora ( ) adelantarHora ( )

miReloj : Reloj
hora = 3 **minuto = 49**

Para el ejemplo, un objeto miReloj es una entidad que tiene un conjunto de propiedades o atributos, como el color, marca, forma, estilo, precio y dentro de los comportamientos están el dar las horas, minutos, adelantarse, atrasarse.

Se tiene dos ejemplos donde se modelan los objetos (computador portátil y teléfono celular) mediante el correspondiente diagrama de objetos.

portatil : Computador
color = "negro" marca = "Lenovo" precio = 1.800.000
prender apagar procesar almacenar

celular : TelefonoMovil
color = "negro" marca = "movil" precio = 180.000
recibirLlamada enviarMensaje recibirMensaje registrarLlamadas

Otros ejemplos de objetos son:

Objetos físicos
- Barcos, aviones, perros, computadores.

Objetos de interfaces gráficas de usuarios:
- Etiquetas, cajas de texto, botones.

Objetos intangibles:
- Cuenta de ahorros, cuenta corriente, CDT.

Ejemplos de no objetos:
- El amor, la felicidad, la nostalgia.

1.8.2.2 Estructura interna de un objeto

Los atributos de una clase definen el estado del objeto, que son los valores de los datos del objeto concreto y que lo diferencian de los demás.

Tabla 3.
Ejemplo estructura interna de un objeto

Objeto	Estado
estudiante	edad = 20 estatura 1.80
celular	color = negro marca = Nokia
computador	marca = Lenovo color = negro

Los métodos de una clase definen el comportamiento del objeto y corresponde a las operaciones que pueden realizarse sobre los objetos de una clase.

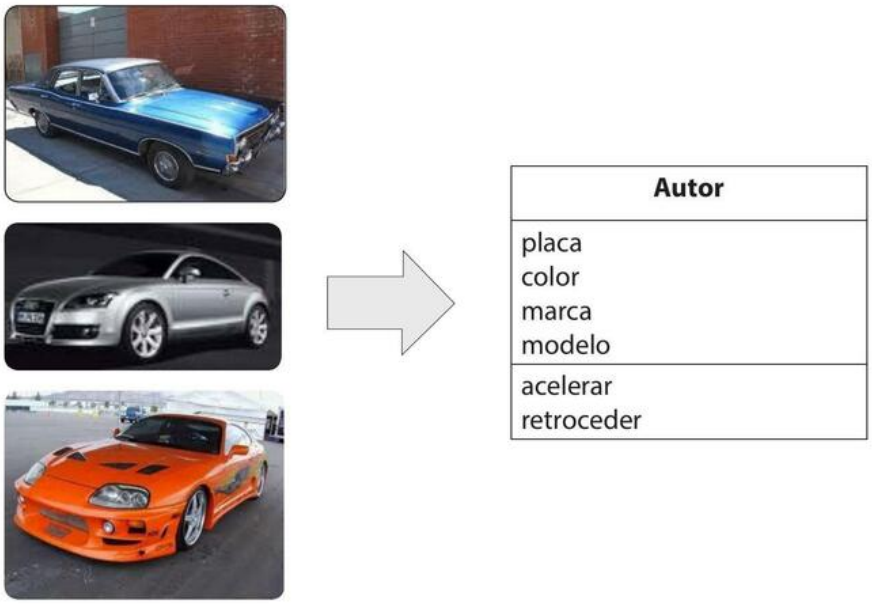
1.9 Visibilidad en atributos y métodos

La programación orientada a objetos ayuda a incrementar factores de calidad como: reutilización, facilidad del uso de las clases, entre otros. Por su parte, la comunicación y visibilidad, tanto para los atributos de una clase como para los métodos, puede ser: *public*, *private*, *protected* (público, privado, protegido).

Público (*public*). Representado por el signo más (+), indica que el método o atributo está visible dentro o fuera de la clase, es decir, es accesible desde todos lados.

Privado (*private*). Representado por el signo menos (-), indica que el atributo o método solamente será accesible dentro de la misma clase (solo sus métodos lo pueden acceder).

Protegido (*protected*). Representado por el símbolo numeral (#), indica que el atributo o método no es accesible desde fuera de la clase, pero se podrá acceder por métodos de la clase, además de las **subclases** que se deriven, como es el caso de la herencia.



Los tres vehículos son objetos que se pueden agrupar en la clase Auto.

1.10 Paquetes

Permiten agrupar los elementos de comportamientos similares que se modelan con UML. La figura 6 representa un diagrama de paquete, similar a una carpeta o *folder*.

Figura 6.
Diagrama de paquete

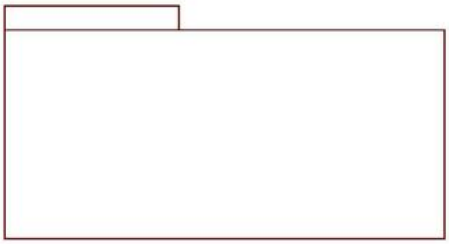
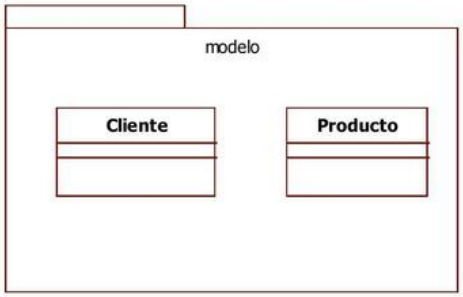


Figura 7.
Diagrama de paquetes con dos clases comunes



La figura 7 ilustra un diagrama de paquetes, conformado por dos clases comunes al modelo de un sistema.

1.11 Lecturas recomendadas

- Qué es UML.
- Historia de UML.
- Paradigma de programación orientada a objetos.
- Atributos, métodos y constructores.
- Principios de la programación orientada a objetos.

1.12 Preguntas revisión de conceptos

1. ¿Dónde utilizar UML?
2. Diferencias entre clases y objetos.
3. ¿Cuáles son los lenguajes que permiten la implementación del concepto de objetos y actualmente son fuente para el desarrollo de proyectos de software?

1.13 Ejercicios

1. Construir la clase llamada Proveedor, que está conformada por los siguientes atributos:

- NIT
- Nombre de la empresa o razón social
- Nombre del proveedor
- Primer apellido del proveedor
- Segundo apellido del proveedor
- Dirección empresa
- Teléfono
- E-mail

2. Crear una clase denominada Cubo, conformada por los siguientes atributos: ancho, alto y largo.

3. Completar la información de la siguiente tabla.

Clase	Objetos
Ropa	camisa El monarca, talla 38; pantalón negro, talla 32
Mamífero	

4. Completar la información de la siguiente tabla:

Clase	Objetos
Mueble	
Gato	
Docente	

1.14 Bibliografía

Aguilar, L. J. (2004). *Fundamentos de programación de algoritmos y estructura de datos*. Tercera edición. Ciudad: México. McGraw-Hill.

Booch, G. (1996). *Análisis y diseño orientado a objetos con aplicaciones*. Ciudad: México. Addison Wesley.

Deitel y Deitel (2012). *Cómo programar en Java*. Ciudad: México. Editorial Pearson (Prentice Hall).

Fowler, M. y Scott, K. (1999). *UML gota a gota*. Ciudad: México. Editorial Pearson.

Villalobos, J. y Casallas, R. (2006). *Fundamentos de programación, aprendizaje activo basado en casos*. Ciudad: México. Editorial Pearson.

Capítulo 2

Componentes de un programa

2.1 Temática a desarrollar

- Identificadores
- Tipos de datos
- Palabras reservadas
- Comentarios
- Variables
- Constantes
- Expresiones
- Estructura de un programa en Java
- Entrada y salida de datos
- Operadores aritméticos

2.2 Introducción

En este capítulo se incluyen definiciones y conceptos relacionados con los tipos de datos en Java, temáticas referentes a la definición de variables, su inicialización, los operadores aritméticos y las palabras reservadas.

Se indican los pasos para la creación de un programa o un proyecto, como la digitación, la compilación y la ejecución de un programa en Java, conceptos básicos de importancia dentro del desarrollo de programas y aplicaciones a utilizarse en la programación.

Como buenas prácticas, se recomienda tener en cuenta la manera de nombrar las clases, los atributos o variables, los métodos y las constantes.

2.3 Características de un programa en Java

- El nombre de la clase debe ser igual al nombre del archivo.
- El nombre del archivo tiene extensión java.
- Los comentarios se colocan entre /* y */ o se inician con //.
- La clase se encierra entre { y }.
- Los métodos también se encierran entre { y }, como el método main.
- Las sentencias (acciones) se terminan con punto y coma.
- Los argumentos de los métodos se colocan entre paréntesis redondo ( ).
- Se usa la indentación, sangrado o tabulación para obtener un código más legible empleando cuatro espacios como unidad de tabulación.
- Se debe evitar las líneas de más de 80 caracteres.

2.4 Estructura de un programa en Java

Java es un lenguaje orientado a objetos. Así que un programa en este lenguaje es una colección de clases, donde cada clase es en un archivo diferente y el nombre del archivo debe ser el mismo que el de la clase.

El programa más sencillo utiliza una clase e incluye el método main( ) (es el método de inicio para la ejecución de las aplicaciones en Java).

Un programa Java consta de una colección de archivos o unidades de compilación, que se describen en los apartados siguientes.

2.4.1 Declaración de importaciones (import)

Nombra un elemento de otro paquete que se utilizará en las declaraciones posteriores de interfaces o clases. Se puede utilizar un asterisco para incluir todos los elementos de un paquete.

- import nombrePaquete.*;
- import nombrePaquete.NombreClase;
- import java.util.Scanner;

2.4.2 Definición de clases

Una clase consta de una declaración y un cuerpo. El cuerpo contiene campos de datos y declaraciones de métodos.

Sintaxis:

```
public class <NombreClase> {
// Declaración de atributos
<tipo> <variable> ;
// Declaración de métodos
<tipo> <nombreMétodo> ( <argumentos> )
{ ... }
}
```

Ejemplo:

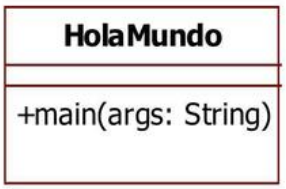
```
public class ClaseUno  {
   // atributos y declaraciones de métodos
}
```

```
public class Fecha {
// Atributos
int dia ;
int mes ;
int año ;
// Métodos
void asignarDia ( int d ) {...}
String darFormato ( ) {...}
}
```

Ejemplo:

Programa que imprime el mensaje "Hola Mundo, ¿cómo estás?" en pantalla.

Diagrama de clases del programa Hola Mundo:



Un diagrama de clases en UML permite especificar la estructura de un sistema en función de las clases que lo componen.

```
public class  HolaMundo {

        /** este es el método principal */
        public static void main(String[ ] args) {
                System.out.println("Hola Mundo, ¿cómo estás?");
        }
}
```

public es una palabra reservada que indica que la clase o el método pueden ser usados por cualquier otra clase.

class es la palabra que indica que a continuación de ella se va a declarar una clase.

Toda clase en Java que sea una aplicación debe tener un método main( ), ya que es la entrada para la ejecución del programa.

{ } permite delimitar el alcance de la clase, determinando dónde inicia y dónde finaliza.

public class HolaMundo define la clase HolaMundo, que corresponde al nombre del programa, los bloques del programa van entre llaves **{ }** que abren y cierran el bloque del programa, ya sea el de la clase o el del método.

public static void main(String [ ]args) es el método principal o punto de ejecución de las aplicaciones en Java, está demarcado por una llave que abre el método y otra para cerrar.

- public: el método se puede usar desde fuera.

- static: el método pertenece a la clase (no a los objetos de la clase).
- void: indica que el método no retorna nada.
- String[ ] args: es el argumento, datos que se pasan a la operación.

System.out.println("Hola Mundo"); imprime el mensaje en pantalla "Hola Mundo"; se puede observar que al final de la línea hay un punto y coma (;) que forma parte de la sintaxis del programa.

- System: es una clase definida que representa al computador.
- out: es un objeto de la clase System definida que representa la pantalla.
- println: método para poner un texto en la pantalla.

Los métodos de salida print( ) y println( ) son utilizados en los programas cuya salida está basada en texto y la impresión se realiza desde la consola.

2.5 Presentación del programa

- Tabulación o sangrado: margen del texto que se usa para remarcar la estructura del programa.
- Comentarios:
  - de documentación (/** …*/).
  - normales, hasta fin de línea (//).
- Uso del punto y coma(;) para finalizar declaraciones e instrucciones.
- Se utilizan { } para definir los contenidos de clases y métodos.

2.6 Construcción de un programa en Java

La construcción de un programa en Java se realiza en varias etapas**:**

Primera etapa

- Creación del proyecto en cualquier entorno de desarrollo.
- Guardarlo en el medio magnético, con el nombre del programa y la extensión java, como, por ejemplo, HolaMundo.java, Calculadora.java, Cuadrado.java.

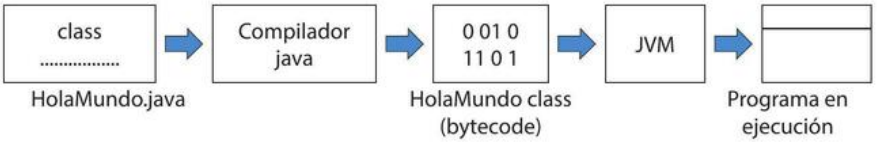
Segunda etapa

- Compilar el programa.

- Corregir errores de sintaxis (expresiones mal digitadas, nombres de variables diferentes a las definidas).
- Se generan el código intermedio bytecode, al compilar el programa creando los archivos punto class, por ejemplo, HolaMundo.class, Calculadora.class, Cuadrado.class.

Tercera etapa

En la etapa de ejecución del programa, la Máquina Virtual de Java (JVM, por sus siglas en inglés) interpreta las instrucciones bytecode, presentando el resultado del programa en pantalla, o realiza alguna tarea, cálculo o proceso.



El bytecode es un lenguaje nativo de cualquier implementación de la máquina virtual de Java. De esta forma, se logra que un programa Java corra en cualquier plataforma.

2.7 Palabras reservadas

Como cualquier lenguaje de programación, Java tiene un conjunto de palabras reservadas incorporado para identificar definiciones, condiciones, ciclos, entre otros términos que no se pueden utilizar como nombres de los identificadores.

En la tabla 4, se relacionan las palabras reservadas:

Tabla 4.
Palabras reservadas en Java

abstract	continue	for	new	switch
boolean	default	goto	null	synchronized
break	do	if	package	this
byte	double	implements	private	threadsafe
byvalue	else	import	protected	throw

case	extends	instanceof	public	transient
catch	false	int	return	true
char	final	interface	short	try
class	finally	long	static	void
const	float	native	super	while

2.8 Los identificadores

Un identificador corresponde a un nombre de variables, constantes, clases, objetos, métodos, los cuales están conformados por una secuencia de letras y dígitos de longitud limitada que permite identificar los elementos de un programa en Java.

Los nombres de los identificadores en Java cumplen algunas reglas, como la de comenzar con una letra, un signo de pesos ($) o el guion bajo (_); no pueden iniciar con un número. Después del primer carácter, los identificadores pueden ir seguidos de cualquier combinación de letras, signo de moneda, números, guion al piso.

2.9 Tipos de datos

En el lenguaje de programación Java, existen dos tipos de categorías de datos: los tipos primitivos, que se caracterizan por tener un único valor, y los tipos de referencia, que se utilizan para referirse o acceder a un objeto. La figura 8 muestra mediante un mapa mental este concepto de tipos de datos.

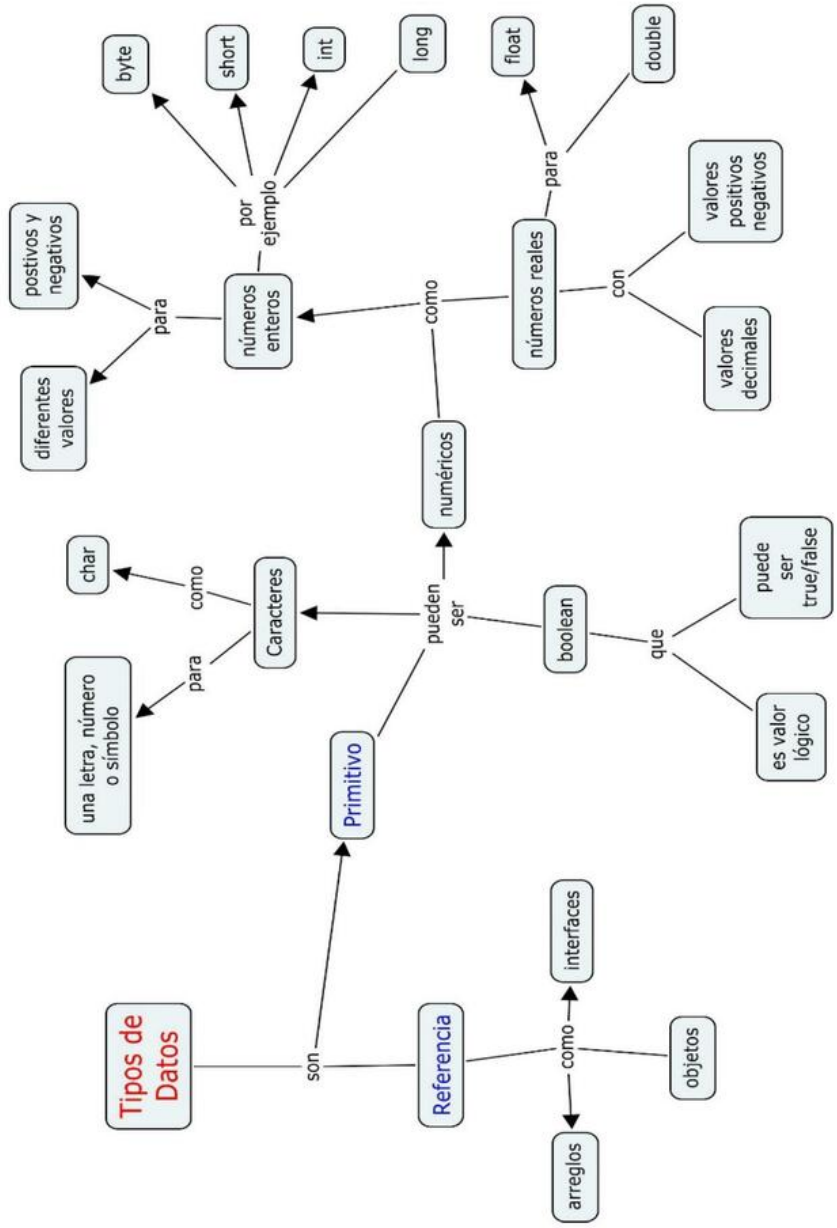


Figura 8.
Mapa mental de tipos de datos

En la tabla 5, se presentan los tipos de datos de Java que se utilizan en los programas de este lenguaje de programación.

Tabla 5.
Tipos de datos de Java

Tipos de datos	Número de bytes	Descripción	Inicialización
boolean	1	Puede contener los valores true o false.	false
byte	2	Enteros. Tamaño 8-bits. Valores entre -128 y 127. (-2^7 ; 2^7-1)	0
short	2	Enteros. Tamaño 16-bits. Entre -32768 y 32767. (-2^{15} ; $2^{15}-1$)	0
Int	8	Enteros. Tamaño 32-bits. Entre -2147483648 y 2147483647. (-2^{31} ; $2^{31}-1$)	0
long	8	Enteros. Tamaño 64-bits. Entre -9223372036854775808 y 9223372036854775807. (-2^{63} ; $2^{63}-1$)	0L
float	8	Números en coma flotante. Tamaño 32-bits.	0.0f
double	8	15 cifras decimales equivalente 4.9E-324 1.7E308	0.2d
char	2	Unicode('\uxxxx')	'\u000'

2.10 Variables

Son los valores que se pueden cambiar durante la ejecución de un programa y corresponden a nombres que representan un valor de cierto tipo int, char, float, double.

Características de las variables

- Es una unidad básica de almacenamiento de valores o datos.
- En Java debe ser declarada antes de utilizarse.
- Cada variable tiene un tipo y un identificador.

- Es un nombre simbólico con un valor.
- Las variables pueden ser iniciadas.
- La forma para declarar las variables es: tipo de dato y nombre de la variable.

Los nombres de las variables

- Deben iniciar con una letra del alfabeto, guion de subrayado.
- Los siguientes caracteres pueden ser dígitos.
- No se pueden usar palabras reservadas como nombres de las variables.
- El estándar para nombrar variables es iniciar con minúscula; para el caso cuando el nombre está compuesto por dos o más palabras, la letra inicial a partir de la segunda palabra va en mayúscula.

costo, iva, x, x1, x2, costoItemFacturado, nombreEmpleado

2.11 Constantes

Una constante es un identificador que es similar a una variable, excepto porque no se le puede asignar un valor en tiempo de ejecución; se escriben en mayúscula sostenida.

Son ejemplos de nombres de constantes: PI, IVA, NOMBRE.

2.12 Comentarios

Se utilizan para hacer descripciones de código y facilitar información adicional que no es legible en el programa. Los comentarios deben contener solo información que es relevante para la lectura y el entendimiento de la aplicación.

En Java hay tres tipos de comentarios empleados para documentar un programa o para cancelar líneas que no sean tenidas en cuenta al momento de compilar el programa.

Comentarios para una sola línea

La pareja de caracteres / / hay que escribirla sin dejar ningún espacio en blanco entre ellos.

```
// Programa que suma dos números
// x variable que almacena un dato de tipo entero
```

Comentarios de bloque

Un comentario de bloque debe ir precedido por una línea en blanco que lo separe del resto del código; estos se usan para dar descripciones del programa utilizado o antes de cada clase, métodos, atributos. También se pueden utilizar al interior de los métodos y deben estar al mismo nivel que el código del método que describen.

```
/*
* Aquí hay un comentario de bloque.
*  comentarios para varias líneas
*/
```

Comentarios de documentación

Describen clases Java, interfaces, constructores, métodos y atributos. Cada comentario de documentación se encierra con los delimitadores de comentarios /**...*/

```
/**
* La clase que modela la información de un cliente
*/
/**
* La clase que modela las transacciones de una cuenta corriente
*/
```

2.13 Definición de variables

Las variables se deben definir antes de utilizarlas o ser nombradas dentro del programa, para lo cual se debe tener en cuenta el tipo de dato y el nombre de la variable.

Tipo de dato	Nombre de variable	
int	suma;	// se declara una variable entera
double	venta;	// se declara una variable double
char	letra	// se declara una variable de tipo char
boolean	respuesta;	// se declara una variable de tipo booleano

Se puede observar que al final de cada sentencia se coloca un punto y coma, que forma parte de la sintaxis de Java, y el doble slash //, que corresponde a comentarios que ayudan a documentar el programa en cuanto a su contenido y funcionalidad.

2.14 Definición de constantes

Se realiza anteponiéndole la palabra final al tipo de dato y una vez inicializadas no se puede cambiar su valor. Las constantes se escriben en mayúscula sostenida todo el nombre.

Tipo de dato	Nombre
final double	PI = 3.1416;
final int	IVA = 16;
final int	VALOR_MAXIMO = 100;
final char	LETRA = 'S';

2.15 Operadores aritméticos

Permiten la realización de las operaciones aritméticas básicas, que trabajan sobre variables numéricas. En la tabla 6 se presentan estos operadores.

Tabla 6.
Operadores aritméticos

Operación	Operador	Ejemplo
suma	+	z=x+y; a=b+2;
resta	-	z=x-y; a=b-2;
producto	*	z=x*y; a=b*2;
división	/	z=x/y; a=b/4;
residuo o módulo de la división	%	z=x%y; a=b%2;
incremento	++	x++;
decremento	--	y--;

```
int a;
int b;
int c;
int d;
a = 2 + 2; // adición
b = a * 3; // multiplicación
c = b - 2; // substracción
d = b / 2; // división
e = b % 2; // resto de la división (módulo)
```

Operadores incremento y decremento

- El operador ++ incrementa en 1 la variable que sigue.
- El operador -- decrementa en 1 la variable que acompaña.

Ejemplos:

```
int x = 3;
x ++; // Se incrementa x en uno; luego queda con el valor de 4;

int y = 3;
y - - // Se decrementa en uno; luego queda con el valor igual a 2;
```

2.16 Los separadores

Los separadores admitidos en Java son:

- Paréntesis **( )**: utilizados para contener listas de parámetros en la definición, llamada a métodos, agrupación de expresiones, contener expresiones para control de flujo.
- Llaves **{ }**: son utilizadas para definir un bloque de código, para clases, métodos, ámbitos locales y para inicializar arreglos en forma automática.
- Corchetes **[ ]**: utilizados para declarar tipos de los arreglos o cuando se referencian valores de los arreglos.
- Punto y coma (**;**): usados para separar sentencias.
- Coma (**,**) permite separar identificadores consecutivos en una declaración de variables, también es utilizada para encadenar sentencias dentro de una sentencia de un ciclo for.
- Punto (**.**): permite separar nombres de paquete de subpaquetes, clases de atributo, objetos de nombre del método.

2.17 Lecturas recomendadas

- Precedencia de los operadores aritméticos.
- Tipos de datos primitivos.
- Tipos de datos referencia.
- Documentación de un programa.

2.18 Preguntas de revisión de conceptos

1. ¿Qué tipo de datos sería recomendable para almacenar los valores de las funciones trigonométricas?

2. ¿Qué es un programa fuente?
3. ¿Qué es compilar un programa?
4. ¿Qué son errores de sintaxis?
5. ¿Qué es el API de Java?

2.19 Ejercicios

1. Definir los tipos de datos para el radio, área y longitud de una circunferencia.
2. Definir cinco tipos de constantes distintas.

2.20 Referencias bibliográficas

Booch, G. (1996). *Análisis y diseño orientado a objetos con aplicaciones*. Ciudad: México. Addison Wesley.

Deitel y Deitel (2012). *Cómo programar en Java*. Ciudad: México. Editorial Pearson (Prentice Hall).

Fowler, M. y Scott, K. (1999). *UML gota a gota*. Ciudad: México. Editorial Pearson.

Rumbaugh, J., Jacobson, I. y Booch, G. (2005). *The Unified Modeling Language User Guide*. Ciudad: San Francisco. Addison-Wesley, Reading, Mass. ISBN-13: 078-5342267976, ISBN-10: 0321267974.

Villalobos, J. y Casallas, R. (2006). *Fundamentos de programación, aprendizaje activo basado en casos*. Ciudad: México. Editorial Pearson.

Capítulo 3

Herramientas para el desarrollo de la programación orientada a objetos

3.1 Temática a desarrollar

- Entornos Integrados de Desarrollo (IDE)
- Herramientas de modelamiento
- El lenguaje de programación Java

3.2 Introducción

La construcción de una aplicación en Java involucra el uso de una serie de herramientas de software que facilitan su modelamiento, codificación, pruebas y su posterior documentación; para ello, hay herramientas case y entornos integrados de desarrollo.

- Los entornos de desarrollo integrados están diseñados para maximizar la productividad del desarrollador de aplicaciones proporcionando componentes unidos con interfaces de usuario similares. Los IDE presentan un único programa en el que se lleva a cabo todo el desarrollo y, por lo general, esta herramienta suele ofrecer muchas características para la creación, modificación, compilación, implementación y depuración de software. Los IDE brindan un marco de trabajo amigable para la mayoría de los lenguajes de programación.

- Las herramientas de modelado UML se convierten en partes fundamentales en el desarrollo de todo sistema de información orientado a objetos, permitiendo representar los requerimientos del sistema, la funcionalidad y la interacción con otros módulos o sistemas.

- Java, como herramienta de programación orientada a objetos, es utilizado para la construcción de las aplicaciones de software, donde los programas se elaboran a partir de módulos independientes, los cuales se pueden modificar o complementar.

El uso adecuado de una herramienta de modelado de software, así como de un entorno integrado de desarrollo y de un lenguaje de programación como Java, son buenas prácticas para el desarrollo de aplicaciones orientadas a objetos.

3.3 Entornos Integrados de Desarrollo (IDE)

Son aplicaciones de software que integran herramientas, componen un entorno de desarrollo integrado por: un editor de texto, un compilador, un intérprete, elementos para la documentación, un depurador, un sistema de ayuda para la construcción de interfaces gráficas de usuario y un generador de control de versiones.

Un Entorno de Desarrollo Integrado (IDE, por sus siglas en inglés) brinda una serie de opciones y funcionalidades para el desarrollo de programas o aplicaciones de software que van desde aplicaciones de escritorio (calculadora, juegos) hasta sistemas de información de tipo empresarial.

Los IDE pueden ser utilizados para un único lenguaje de programación, pero algunas de estas herramientas, como Eclipse o NetBeans, soportan más de un lenguaje de programación, dentro de los que están Java, C++, Ruby o PHP, entre otros, ya que incluyen plugins (programa que incrementa las funcionalidades del IDE).

Algunos de los principales y más populares IDE para Java son:

- BlueJ: desarrollado como un proyecto de investigación universitaria, es libre.
- Eclipse: desarrollado por la Fundación Eclipse, es libre y de código abierto.
- IntelliJ IDEA: desarrollado por JetBrains, es comercial.
- JBuilder: desarrollado por Borland, es comercial, pero también existe la versión gratuita.
- JCreator: desarrollado por Xinox, es comercial, pero también existe la versión gratuita.
- JDeveloper: desarrollado por Oracle Corporation, es gratuito.
- NetBeans: es gratuito y de código abierto.

3.3.1 Eclipse

Entorno de desarrollo integrado de código abierto y multiplataforma (Windows, Linux), es una herramienta para la programación, desarrollo y compilación de aplicaciones en C++, Java, PHP, entre otros.

Se descarga del sitio http://www.eclipse.org/, seleccione la opción "Downloads" o digite directamente http://www.eclipse.org/downloads/ y eleja Eclipse Classic, para 32 o 64 bits, según la arquitectura del computador a trabajar.

Figura 9.
Pantalla de carga de Eclipse LUNA

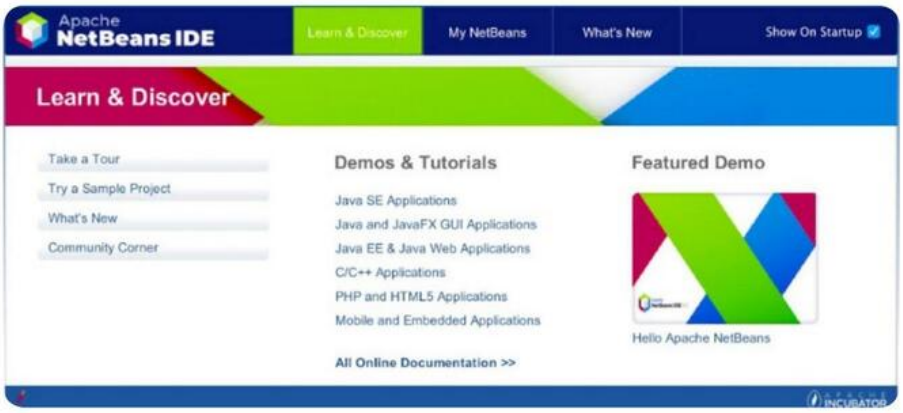
La figura 9 corresponde a la versión LUNA de Eclipse, versión 4.4, cuyo lanzamiento oficial fue el mes de junio del 2014.

3.3.2 NetBeans

Es otro entorno integrado de desarrollo para los programadores que les permite digitar, compilar, depurar y ejecutar programas. Está construido en Java, pero puede servir para cualquier otro lenguaje de programación. NetBeans facilita la construcción de aplicaciones a partir de un conjunto de componentes de software llamados módulos.

La figura 10 corresponde a la pantalla de inicio de la versión lanzada al mercado en noviembre del 2018.

Figura 10.
Pantalla inicio en NetBeans



Se descarga del sitio http://netbeans.org/ o http://netbeans.org/downloads/ y puede seleccionar entre las tecnologías: Java SE, JavaFX, Java, Ruby, C/C++, PHP, All (todo el paquete completo).

3.3.3 BlueJ

Es un entorno integrado desarrollado de un proyecto de investigación académico de las universidades de Deakin University, Melbourne, Australia y la Universidad de Kent, en Canterbury, UK, diseñado para enseñar la programación orientada a objetos a principiantes con especial énfasis en las técnicas de visualización e interacción en un ambiente que fomenta la experimentación y la exploración. Se puede descargar del sitio web http://bluej.org/. La figura 11 corresponde a la pantalla de inicio de BlueJ.

Figura 11.
Pantalla de inicio de BlueJ

3.3.4 GreenFoot

Es un entorno integrado desarrollado especialmente para programadores novatos. Si bien soporta el lenguaje Java en su totalidad, es especialmente útil para programar ejercicios que tengan contenido visual. En GreenFoot, la interacción y visualización de objetos son los elementos fundamentales. El sitio web oficial del proyecto es http://www.greenfoot.org/door.

La figura 12 corresponde a la pantalla de inicio de GreenFoot.

Figura 12.
Pantalla inicio de GreenFoot

3.3.5 IntelliJ IDEA

Es uno de los mejores IDE utilizados para Java, de la mano de JetBrains, que siempre tuvo características que no se encuentran en otros (Eclipse, NetBeans, entre otros), como el guardado y compilado automático, control de sugerencias y resaltado de código dentro incluso de cadenas, el más inteligente autocompletado de código y mucho más. La versión 15 salió el mes de noviembre de 2015 y trae varias novedades y características especiales para los desarrolladores Android. En esta publicación se citan las nuevas características y lenguajes, así como tecnologías frameworks soportadas.

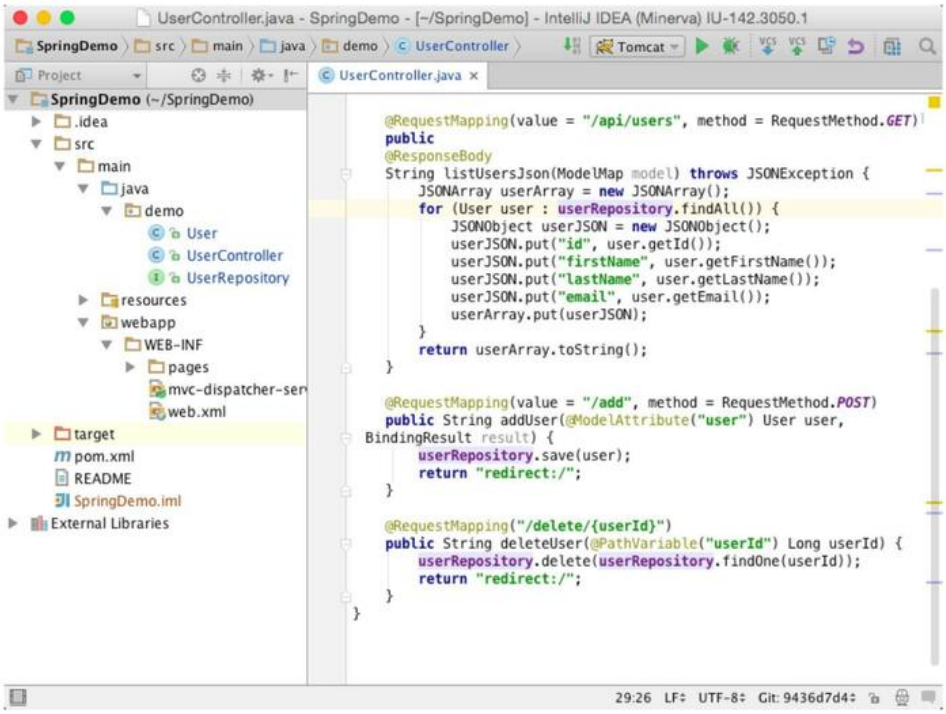
Características:

- Es multiplataforma (Windows, Mac OS X, Linux).
- Existen dos ediciones: Comunity Edition, que es de código abierto, y Ultimate, de pago.

- Mejor compilación, rendimiento mejorado que redujo requisitos de la memoria.
- Soporte para Java 8, la nueva generación de la plataforma Java y con su nueva sintaxis de las expresiones lambda.
- Actualización de las herramientas del framework Spring, con mejoras en rendimiento y asistencia.
- Entre los lenguajes soportados están: Java, Groovy, XML, Regexp, Scala, Clojure, Ruby/JRuby, Python, PHP, entre otros.
- Sitio oficial de descarga: https://www.jetbrains.com/idea/.

La siguiente imagen visualiza el entorno de JetBrains con una aplicación en Java.

Figura 13.
Pantalla IntelliJ IDEA



3.4 Herramientas de modelamiento

UML es un lenguaje visual para modelar las partes esenciales de un sistema de software. Es una herramienta útil para las etapas de diseño e implementación de aplicaciones de software y también para la documentación de un proyecto de software.

El desarrollo de aplicaciones de software puede llegar a ser un proceso complejo y a menudo difícil que, en algunos casos, requiere la integración de muchos sistemas. Desde el modelado y diseño hasta el código, administración del proyecto, pruebas, despliegue, administración de cambios y más, el modelado basado en UML se ha convertido en una parte esencial para administrar esa complejidad.

3.4.1 StarUML

Es una herramienta para el modelamiento de software basado en los estánda res UML (*Unified Modeling Language*) y MDA (*Model Driven Arquitecture*), el cual permite:

- Soporte para el diseño de diagramas UML.
- Construir los planos para una aplicación de software.
- Generar código a partir de los diagramas y viceversa, actualmente funcionando para los lenguajes C++, C# y Java.

3.4.2 Dia

Programa que está desarrollado para la construcción de diagramas UML. Como todo producto de software libre, es gratuito y de código abierto.

Dia para Linux presenta una interfaz amigable de fácil manejo para elaborar diferentes tipos de diagramas, como los diagramas de UML, diagramas de flujo, diagramas de red.

3.4.3 ArgoUML

Es una herramienta utilizada en el modelaje de sistemas, mediante la cual se realizan diseños en UML (*Unified Modeling Language*) llevados a cabo en el análisis y diseño de sistemas de software.

3.4.4 Rational Software Modeler

Es una herramienta de diseño y modelado visual basada en UML para la documentación y la comunicación con diferentes vistas de un sistema. Debe ser adquirida la licencia para poder utilizarlo.

3.5 Java como lenguaje de programación

Un lenguaje de programación está construido a partir de un conjunto de expresiones, símbolos y reglas sintácticas y gramaticales que permiten escribir instrucciones o sentencias válidas para ser ejecutadas en un computador.

Dentro de los lenguajes de programación orientada a objetos están C++, Java, C#, PHP a partir de la versión 5.0, entre otros. Para efectos del desarrollo de proyectos y la codificación de los programas de este documento, se utilizará Java como lenguaje de programación.

Java es un lenguaje de programación orientado a objetos desarrollado por Sun Microsystems al iniciar la década de 1990; posteriormente fue adquirido por Oracle. La tecnología Java está compuesta básicamente por dos elementos: el lenguaje Java y su plataforma, que corresponde a la máquina virtual de Java (*Java Virtual Machine*, JVM).

La figura 14 detalla, mediante un mapa mental, los fundamentos del lenguaje de programación Java y la figura 15 describe las plataformas.

Java permite el desarrollo de aplicaciones o soluciones para:

- Desarrollo de aplicaciones de software según necesidades.
- Desarrollo empresarial.
- Dispositivos móviles.

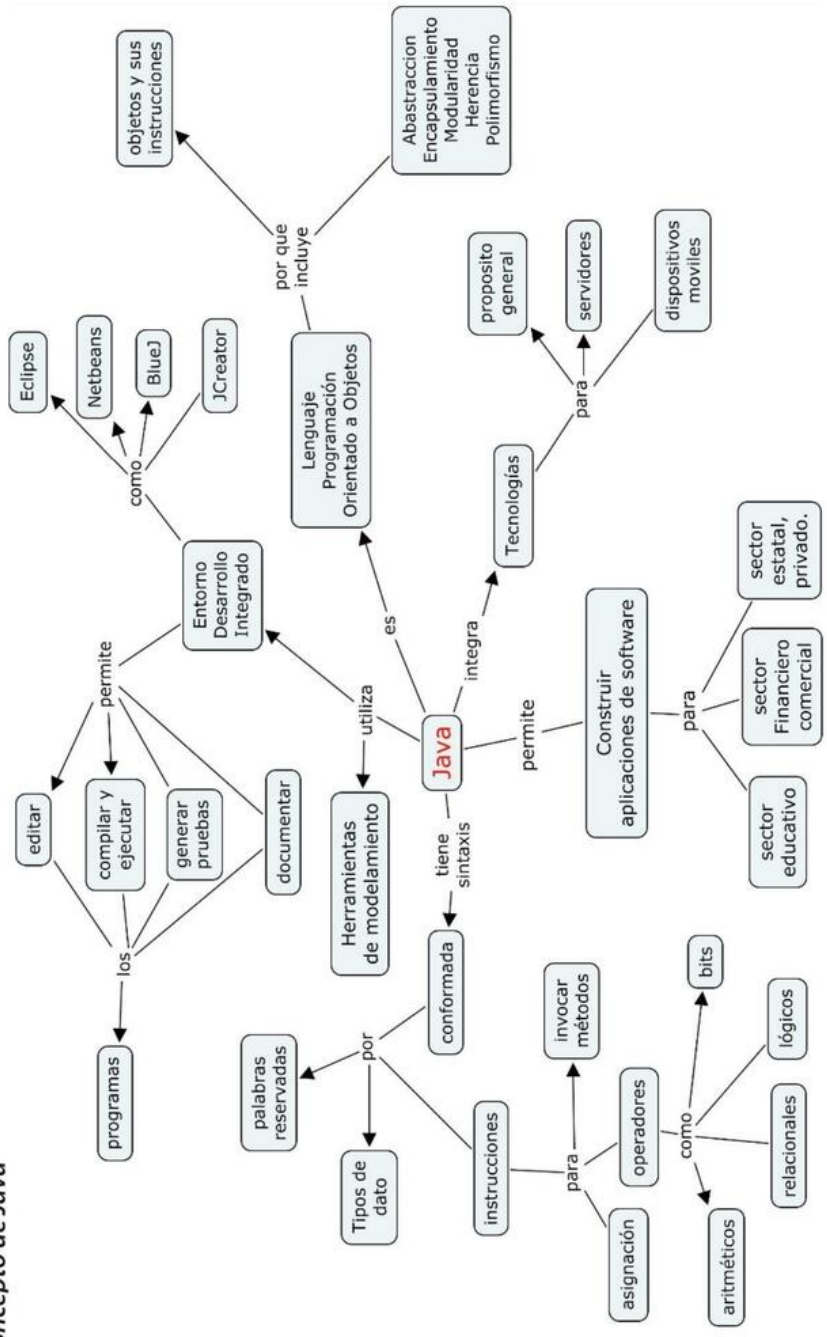
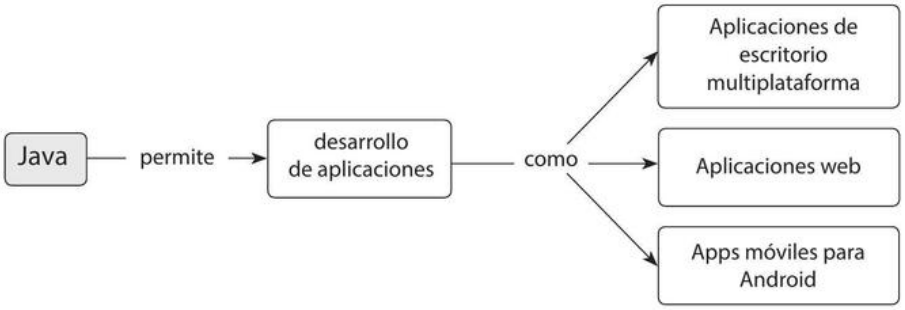


Figura 14.
Concepto de Java

Figura 15.
Concepto plataformas de Java



Algunas de las características de Java son:

- Una misma aplicación puede funcionar en diversos tipos de computadores y sistemas operativos: Windows, Linux, Solaris, MacOS-X, así como en otros dispositivos inteligentes.
- Los programas Java pueden ser aplicaciones independientes.
- Su desarrollo está impulsado por un amplio colectivo de empresas y organizaciones, integrado con la filosofía de software abierto y entorno colaborativo.

3.6 Lecturas recomendadas

- ¿Qué son los entornos de desarrollo integrados?
- ¿Qué son Eclipse, NetBeans y BlueJ? Mencionar sus principales características.
- Importancia del modelamiento.
- Fundamentos, versiones y evolución de Java.

3.7 Preguntas de revisión de conceptos

- Importancia de los IDE.
- Diferencias entre IDE.
- ¿Qué son las variables de entorno?

3.8 Ejercicios

1. Descargar Java del sitio web oficial e instalarlo.
2. Descargar los entornos de desarrollo integrado:
   - Eclipse
   - NetBeans
   - JBlue
   - IntelliJ IDEA
3. Ejecutar cada uno de los anteriores entornos integrados de desarrollo.
4. Descargar las herramientas de diseño:
   - Staruml
   - Argouml
   - Dia

3.9 Referencias bibliográficas

Las siguientes referencias corresponden a los sitios web donde se puede obtener información complementaria de los temas tratados en este capítulo, así como la descarga de los IDE:

https://netbeans.org/
https://eclipse.org/
http://www.bluej.org/
https://www.jetbrains.com/idea/

Capítulo 4

Métodos
Métodos constructores, set y get

4.1 Temática a desarrollar

- Métodos constructores
- Métodos set
- Métodos get
- Modificadores
- Requerimientos
- Entrada y salida de datos
- Strem
- Clase entrada y salida de datos

4.2 Introducción

En este capítulo se incluye, como primer tema, el concepto de constructor, el cual es un método empleado para inicializar valores en instancias de los objetos. La principal característica de los constructores es que llevan el mismo nombre de la clase y no retornan ningún tipo de dato.

El segundo tema a tratar y que se quiere resaltar son los métodos accesores y modificadores (set/get). El método set es utilizado para actualizar el valor de una variable y el método get para obtener el valor de la variable, son la forma de acceder a los atributos de la clase definidos como tipo privado, haciendo necesario implementar los métodos setters y getters para cada atributo de la clase definida; su implementación corresponde a una buena práctica de programación o técnicas para desarrollar y mantener software de calidad.

La temática se complementa mediante ejercicios prácticos, haciendo uso de las diferentes formas para la entrada y salida de datos estándar.

4.3 Métodos

Corresponden a la lógica de la clase; son series de instrucciones que realizan alguna tarea en específico. En las clases, se pueden identificar diferentes tipos de métodos:

- Métodos inicializadores: inicializan los atributos de la clase.
- Métodos selectores: devuelven el valor de los atributos.
- Métodos modificadores: permiten cambiar el valor de los atributos.
- Métodos visualizadores: imprimen el objeto, es decir, el valor de los atributos.
- Métodos operadores: realizan cálculos y generan resultados.
- Constructores: inicializan los atributos del objeto.

La figura 16 ilustra mediante un mapa mental este concepto de método.

4.3.1 Método constructor

Es un método más de la clase que presenta las siguientes características:

- Un constructor es un método miembro público con el mismo nombre de la clase.
- No está indicado ningún tipo de retorno int, double, String, ni siquiera void.
- El objetivo del constructor es reservar memoria e inicializar el valor de las variables miembro de la clase.
- Es un método que se llama automáticamente cada vez que se crea un objeto de una clase.

La figura 17 detalla el concepto de método constructor.

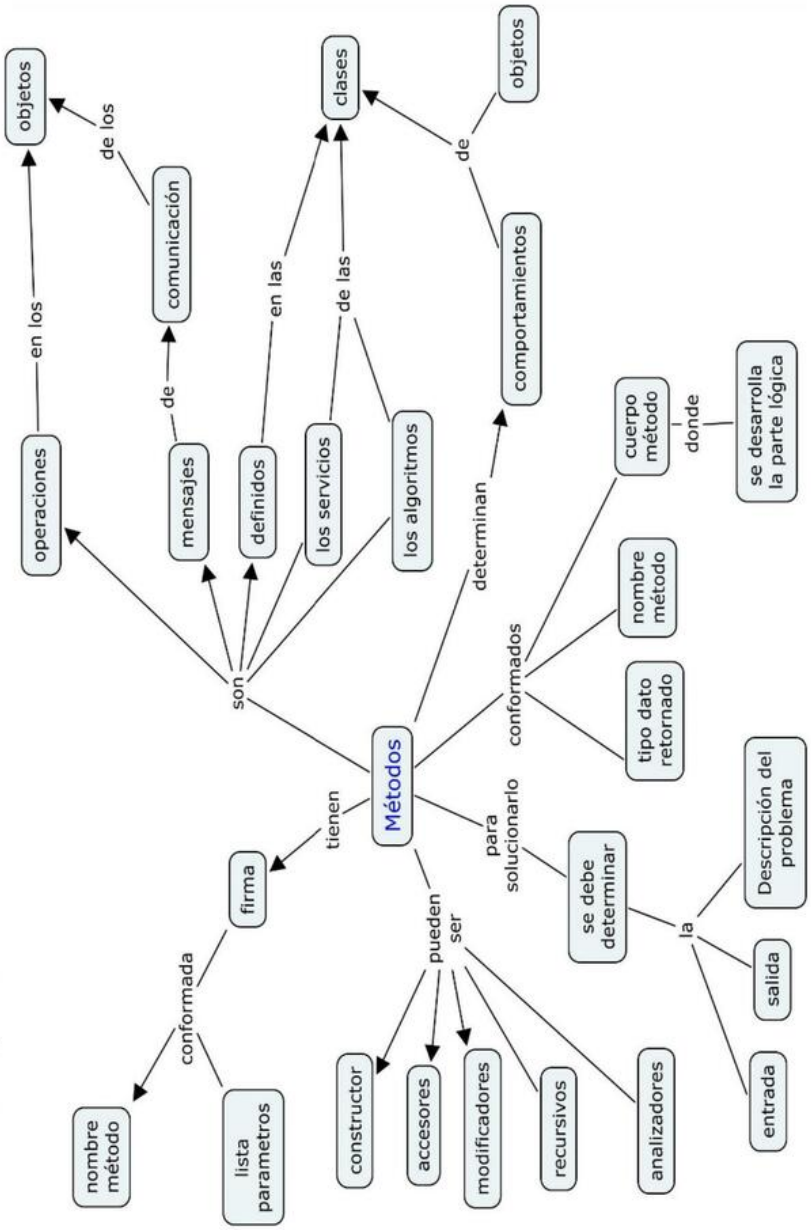


Figura 16.
Mapa mental del concepto de métodos

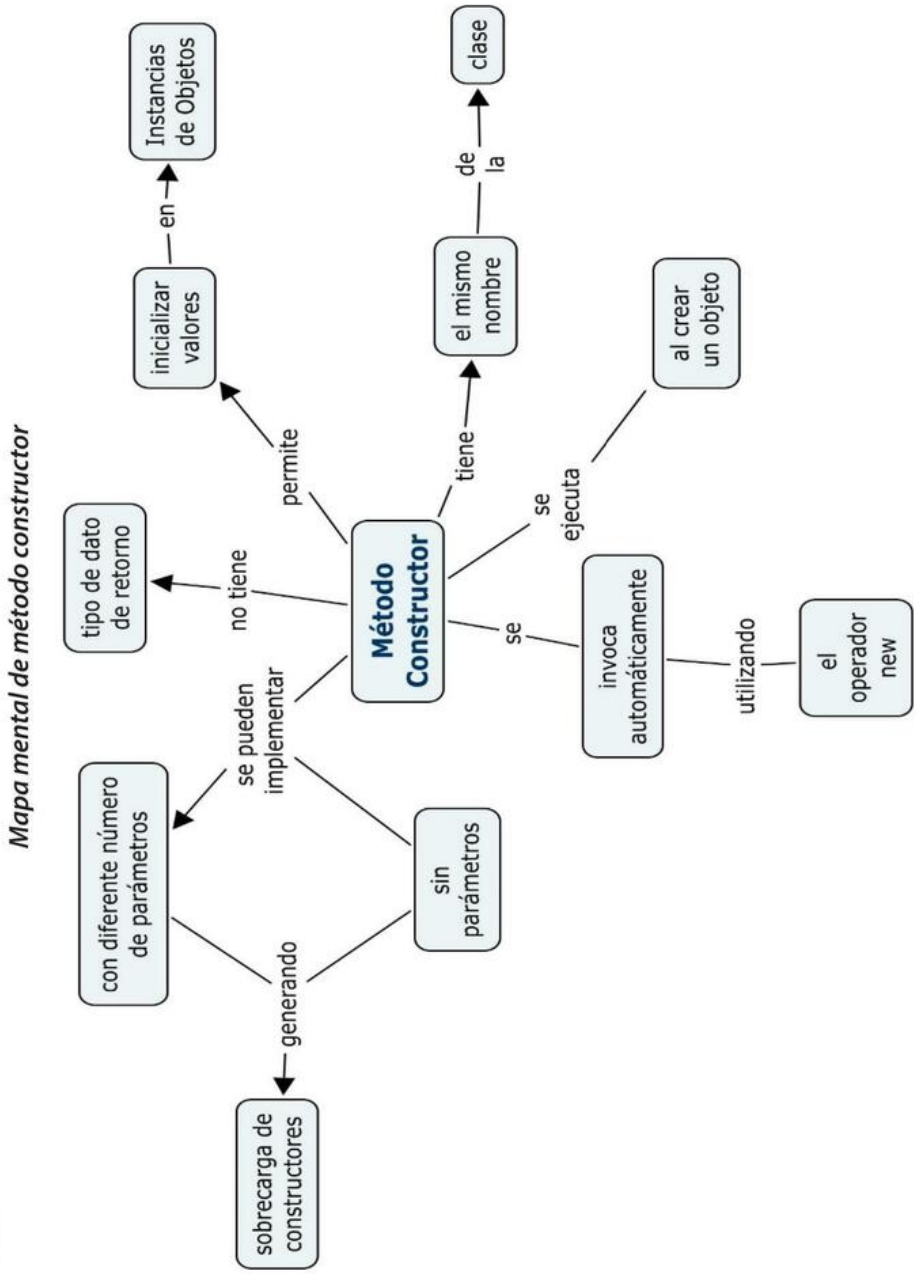

Mapa mental de método constructor
Método Constructor
no tiene
tipo de dato de retorno
permite
inicializar valores
en
Instancias de Objetos
tiene
el mismo nombre
de la
clase
se ejecuta
al crear un objeto
se
invoca automáticamente
utilizando
el operador new
se pueden implementar
con diferente número de parámetros
sin parámetros
generando
sobrecarga de constructores

Figura 17.

Definición de los métodos constructores

La sintaxis para la definición de los constructores es como la de cualquier otro método, conformado por el mismo nombre de la clase, una lista de parámetros separados por coma y no puede haber ningún otro método que comparta el nombre de la clase.

```
class <NombreClase>
{
...
public <NombreClase> ( ) { }
...
}

class Calculadora
{
...
public Calculadora ( ) { }
...
}
```

Llamada a un constructor

Al constructor de una clase se le invoca en el momento de crear el objeto utilizando el operador new. La forma de uso de este operador es:

new <llamada al constructor>
new unaCalculadora( );

Por ejemplo: Calculadora unaCalculadora = new Calculadora();

new es una palabra reservada que se utiliza para construir un objeto de una clase, permitiendo ejecutar el constructor e inicializando los campos del objeto.

Calculadora	unaCalculadora	= new	Calculadora(x,y)
Clase	**Objeto**		**Constructor**

4.4 Modificadores de acceso

Los modificadores son elementos del lenguaje que se colocan delante de la definición de los atributos, métodos o clases, los cuales alteran o condicionan el significado del elemento y brindan la posibilidad de poder establecer la visibilidad o manera de dar los permisos que tendrán otros objetos para acceder a los métodos y/o atributos de la clase.

Los modificadores de acceso permiten al diseñador de una clase determinar quién puede acceder a los datos y métodos miembros de una clase. Tiene la siguiente representación sintáctica.

[modificadores] tipo_variable nombre;
[modificadores] tipo_devuelto nombre_Metodo (lista_Argumentos);

Modificador public
Aplicado a un atributo o método, significa que estos pueden ser vistos y utilizados por objetos de clases siendo accesibles por cualquier referencia (objeto) que los invoque. Se representa con el signo más (+).

Modificador private
Es el nivel más restrictivo de todos los modificadores, se puede utilizar en atributos como en métodos, solo se puede acceder al elemento desde métodos de la clase o solo puede invocarse el método desde otro método de la clase. Se representa con el signo menos (-).

this
Palabra reservada para acceder a variables de instancia de una clase, this se utiliza siempre que se quiere hacer referencia al objeto actual de la clase o cuando hay ambigüedad entre los nombres de los campos del objeto.

La palabra reservada this permite especificar que la variable que señala (this.x) es de la misma clase de la que se usa.

```
public class Punto {
        /**
         * atributos
         */
```

```
        private double x;
        private double y;

        /**
         * constructor con dos parámetros
         * @param x
         * @param y
         */

        public Punto (double x, double y) {
                this.x = x;
                this.y = y;
        }
}
```

this. x = x;

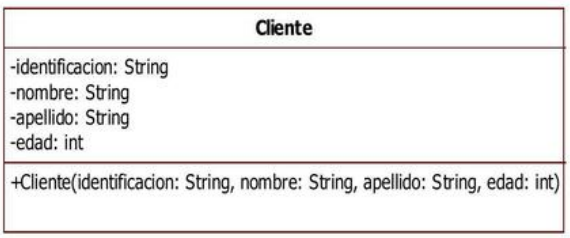
- La primera x corresponde al atributo de la clase.
- La segunda x es el parámetro del método (double x).

El this evita la ambigüedad, ya que el nombre del atributo y del parámetro son el mismo (x).

Ejemplos:

1. La clase que modela la identificación, el nombre, el apellido, la dirección atributos correspondientes a la información del cliente.

Diagrama de clases:



El signo "-" delante de las **propiedades** identificación, nombre y apellido y edad indica acceso privado.
El signo "+" delante de los **métodos** indica acceso público.

Codificación de la clase Cliente:

Es una buena práctica en la programación orientada a objetos declarar siempre todas las variables con el identificador de acceso privado "private", ya que esta variable solo será accesible desde la clase, con lo cual se está brindando la seguridad del programa.

```
public class Cliente {
        private String identificacion;
        private String nombre;
        private String apellido;
        private int edad;
/**
        * constructor de la clase, con cuatro parámetros:
   *identificacion, nombre,
        * apellido, edad
        */
        public Cliente(String identificacion,  String nombre, String apellido, int
edad)  {

                this.identificacion = identificacion;
                this.nombre = nombre;
                this.apellido = apellido;
                this.edad = edad;
        }
}
```

2. Clase que modela los datos de la información de un auto conformada por la placa, la marca, el modelo, el color, siendo estos los atributos del auto.

Diagrama de clases:

Auto
-placa: String -marca: String -modelo: String -color: String
+Auto(placa: String, marca: String, modelo: String, color: String)

Codificación de la clase Auto:

```
public class Auto {
        private String placa;
        private String marca;
        private String modelo;
        private String color;

/**
        * constructor con los parámetros placa, marca,
    * modelo, color
        */
        public Auto(String placa, String marca, String modelo, String
         color) {
                this.placa = placa;
                this.marca = marca;
                this.modelo = modelo;
                this.color = color;
        }
}
```

Una clase puede definir varios constructores con distintos tipos de argumentos, lo que se conoce como sobrecarga de constructores.

3. Se construye una aplicación con dos clases: la clase Cliente, con dos constructores, el primero sin parámetro y el segundo con una lista de parámetro, separados por comas, y la segunda clase denominada PruebaCliente, se crean dos objetos de la clase Cliente.

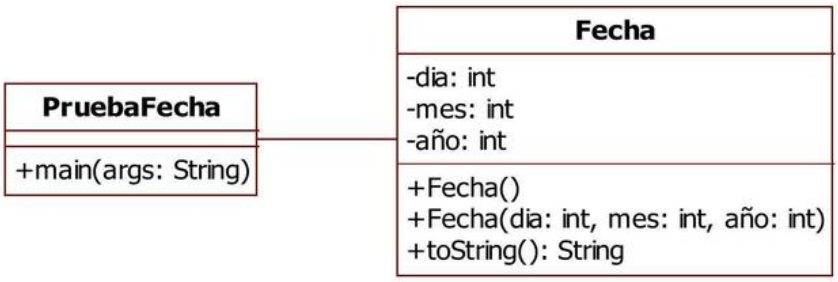
```
public class Cliente {
        private String identificacion;
        private String nombre;
        private String apellido;
        private int edad;
        /**
        * primer constructor de la clase los atributos se
    *incializan con los valores por defecto
        */
        public Cliente() {
                this.identificacion = "";
```

```
                this.nombre = "";
                this.apellido = "";
                this.edad = 0;
        }
        /**
         * Segundo constructor de la clase, con cuatro
         * parámetros identificación, nombre, apellido, edad
         */
        public Cliente(String identificacion, String nombre, String
           apellido, int edad) {
                this.identificacion = identificacion;
                this.nombre = nombre;
                this.apellido = apellido;
                this.edad = edad;
        }
}
public class PruebaCliente {
        public static void main(String[] args) {
           Cliente unCliente = new Cliente();
           Cliente otroCliente = new Cliente("123456", "Alan ",
                        "Brito", 30);
        }
}
```

Cliente unCliente = new Cliente();: se crea el objeto unCliente y se invoca al constructor sin parámetros.

Cliente otroCliente = new Cliente("123456", "Alan ", "Brito", 30);: se crea el objeto otroCliente y se invoca el constructor con parámetros, el cual inicializa los valores del objeto.

4. Aplicación fecha diseñada con dos clases: la clase Fecha, conformada por los atributos día, mes y año, en la que se implementan dos constructores, el primero sin parámetros (los atributos son inicializados por defecto) y el segundo con tres parámetros (día, mes, año); la segunda clase se denomina PruebaFecha.




```
public class Fecha {
        private int dia;
        private int mes;
        private int año;
        /**
         * Primer Constructor
         * Asigna los valores 30, 7 y 2015 a los atributos
         * día, mes y año respectivamente
         */
        public Fecha( ) {
                this.dia = 30;
                this.mes = 7;
                this.año = 2015;
        }
/**
         * Segundo Constructor
         * @param dia, el día del mes a almacenar
         * @param mes, el mes del año a almacenar
         * @param año, el año a almacenar
         */

public Fecha(int dia, int mes, int año) {
                this.dia = dia;
                this.mes = mes;
                this.año = año;
        }
        /**
         * Método que retorna el objeto de la fecha
         */
        public String toString() {
                return ""+dia + "/" + mes + "/" + año;
```

```
        }
}

public class PruebaFecha {
        public static void main(String[] args) {
                Fecha unaFecha = new Fecha();
                Fecha miFecha = new Fecha(6,1,1995);
                System.out.println("La Fecha del objeto
"+miFecha.toString());
        }
}
```

En la clase PruebaFecha se definen dos objetos: unaFecha y miFecha, este último con los tres parámetros, correspondientes al día, mes y año.

El objeto miFecha invoca al método toString(), el cual retorna la fecha, el dato almacenado, el objeto, que es: 6/1/1995.

4.5 Métodos set y get

Los objetos encapsulan (estado y comportamiento) el estado definido por los datos o atributos del objeto y el comportamiento u operación del objeto definido por los métodos.

Con la finalidad de crear una puerta de entrada y de salida para modificar y obtener los atributos del objeto, se implementan los métodos set y get respectivamente para cada atributo de la clase.

Los métodos set se utilizan para inicializar o cambiar el valor de los atributos; set, al traducirlo, podría corresponder a establecer, actualizar, siendo un método que cambia el valor del atributo, ya que este método tiene parámetro.

Los métodos get se emplean cuando se requiere obtener el valor de algún atributo; get, al traducirlo, es conseguir, obtener, y retorna el valor del atributo, ya que por lo general no tiene ningún parámetro.

La figura 18 representa mediante un mapa mental los conceptos relacionados con los métodos set y get.

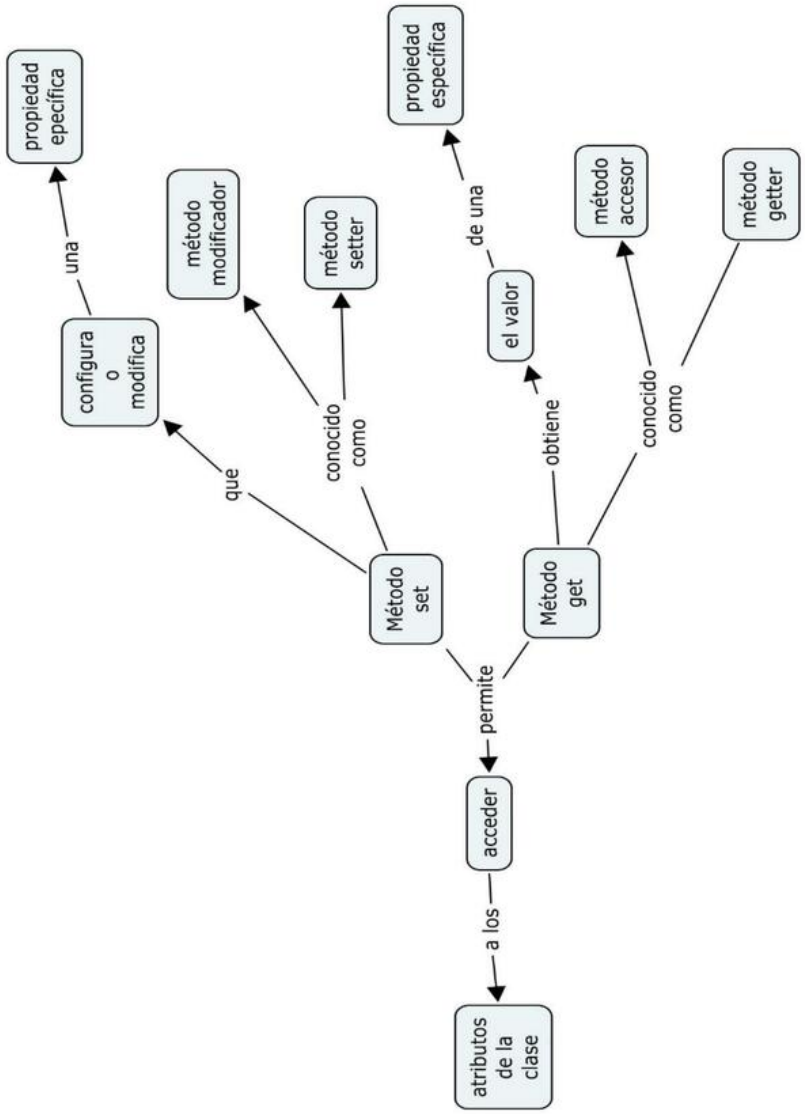


Figura 18.
Mapa mental de los métodos set y get

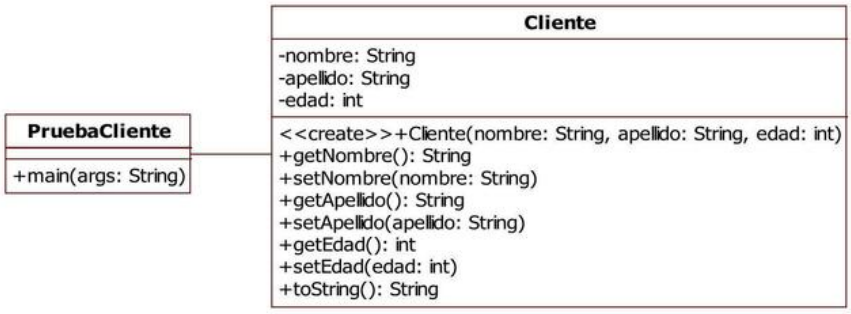
Ejemplos:

1. De la clase que modela el cliente, se ha definido el atributo nombre, para el cual se implementan los métodos: setNombre, que tiene como parámetro el nuevo dato a cambiar en el atributo, y getNombre, que retorna el valor actual del atributo nombre.

```
public class Cliente {
        private String nombre;
        public String getNombre() {
                return nombre;
        }
        public void setNombre(String nombre) {
                this.nombre = nombre;
        }
}
```

Aplicación conformada por la clase Cliente en la que se definen los atributos nombre, apellidos y edad; además, se implementa un constructor y los métodos set y get para cada atributo. En la segunda clase, denominada Prueba Cliente, se define el objeto, unCliente de la clase Cliente.

Diagrama de clases:



Para garantizar la reutilización de las clases, es una buena práctica de programación crear las clases en archivos diferentes.

```
public class PruebaCliente {
        public static void main(String[] args) {
                Cliente unCliente = new Cliente("Oscar","Hernández",12);
```

```
                //impresión de los datos del objeto al ser definido
                System.out.println("Datos Cliente "+unCliente.toString());
                //se cambian los datos del objeto unCliente
                unCliente.setNombre("Mario");
                unCliente.setApellido("Cabrera");
                unCliente.setEdad(41);

                // se imprimen los datos en forma individual del objeto
                System.out.println("Nombre: "+unCliente.getNombre());
                System.out.println("Apellidos: "+unCliente.getApellido());
                System.out.println("Edad: "+unCliente.getEdad());

                //impresión de los datos del objeto
                System.out.println("Datos del cliente
"+unCliente.toString());

        }
}

/**
 * Clase que modela los datos del cliente
 *
 */
public class Cliente {
        private String nombre;
        private String apellido;
        private int edad;

        /**
         * constructor de la clase, con tres parámetros: nombre, apellido, edad
         */

        public Cliente(String nombre, String apellido, int edad) {
                this.nombre = nombre;
                this.apellido = apellido;
                this.edad = edad;
        }


        /**
         * método que retorna el nombre
```

```
 * @return nombre
 */
public String getNombre() {
        return nombre;
}
        /**
 * método que actualiza o cambia el nombre
 * @param nombre
 */
public void setNombre(String nombre) {
        this.nombre = nombre;
}

/**
 * método que retorna el apellido
 * @return nombre
 */
public String getApellido() {
        return apellido;
}

/**
 * método que actualiza el apellido
 * @param apellido
 */
public void setApellido(String apellido) {
        this.apellido = apellido;
}
/**
 * método que retorna la edad
 * @return edad
 */

public int getEdad() {
        return edad;
}

/**
 * método que actualiza o cambia la edad
 * @param edad
 */
```

```
        public void setEdad(int edad) {
                this.edad = edad;
        }

        /**
         * método que retorna el nombre, apellido y edad
         *
         */
        public String toString() {
                return nombre+""+apellido+""+edad;
        }
}
```

4.6 Métodos de instancias

Un método proporciona la implementación del comportamiento dinámico de los objetos y puede cambiar el estado de los objetos que llama. También son los algoritmos de las clases, correspondientes a las operaciones que puede realizar un objeto, siendo la mejor forma de crear y mantener un programa, construido por piezas o módulos denominados métodos.

Los métodos se declaran en las clases y su implementación está integrada por el tipo de retorno, el nombre y un listado de parámetros que pueden expresarse así:

```
tipoRetorno nombreMétodo( [lista_de_argumentos] ) {
   cuerpoMétodo
}
```

Existen diferentes tipos de métodos, dentro de los que están: constructores, accesores y modificadores (set/get).

En Java, la manera de representar las entradas y las salidas es por medio de los streams o flujos de datos, que corresponden a una conexión entre el programa y la fuente o destino de los datos, para lo cual Java tiene implementada una serie de clases, dentro de las que están: Scanner, BufferedReader, entre otras.

4.7 Estructura de un método

Los programas en Java se construyen mediante clases, las cuales están conformadas por atributos y métodos, y a partir de estas se pueden crear los objetos.

Los métodos son los algoritmos o servicios de las clases y corresponden a la parte lógica de la programación.

Está conformado por la declaración y el cuerpo del método.

```
Declaración del método() {
   Cuerpo del método
}
```

La declaración o firma del método está conformada por el tipo de retorno, el nombre, que debe ser una expresión verbal que indica una acción, y una lista de parámetros o argumentos (opcional).

El cuerpo del método está construido por las sentencias que van a realizar una tarea específica, como, por ejemplo, calcular el valor del IVA de un producto, multiplicar dos números, obtener el promedio de ventas al año, ente otros.

La sintaxis general de un método se puede definir así:

```
Tipo dato devuelto  nombre del método (lista argumentos o parámetros) {
     Bloque o conjunto de instrucciones;
}
```

El tipo de dato devuelto: puede ser int, long, float, double, char, String, entre otros, dependiendo del requerimiento.

Nombre del método: se recomienda que sea un verbo o acción verbal, como, por ejemplo: sumar, obtenerSuma, calcularVentas, validarFecha. Cuando el nombre del método está conformado por nombres compuestos, el primer carácter de la segunda palabra inicia con mayúscula y así sucesivamente.

Lista de argumentos o parámetros: se expresa declarando el tipo de dato y el nombre de este, separado por comas.

Los métodos se pueden considerar como máquinas que realizan tareas específicas, los cuales pueden tener una o más entradas y una salida.

Ejemplos:

1. Se requiere sumar dos números. Para realizar este proceso, se deben ingresar dos valores, por ejemplo, 9 y 5; la salida generada es 14, como se puede observar en el siguiente gráfico:

2. Para calcular el cubo de un número, se necesita un valor de entrada, por ejemplo, 3; la salida generada es 27, como se puede observar a continuación:

3. Implementar un método que permita obtener el producto de dos números. Lo primero que se debe hacer para el desarrollo y construcción del método es:

- Realizar la lectura del enunciado del problema y comprender cuál es el requerimiento.
- Determinar entradas y salidas.
- A partir de las entradas, se realizan los correspondientes procesos de forma lógica para obtener el resultado.
- Retornar resultado si es necesario.

4.8 Prueba de escritorio

Una prueba de escritorio es la comprobación lógica de un algoritmo (método) y que brinda una alternativa de solución al problema.

Para desarrollar la prueba de escritorio, se recomienda realizar el siguiente proceso:

- Tomar datos específicos como entrada y seguir la secuencia indicada en el método hasta obtener un resultado.

- Al realizar el análisis del o de los resultados de la prueba de escritorio, se observa si estos son óptimos, lo que quiere decir que el método está correcto; en caso contrario, se tendrán que realizar los correspondientes ajustes.

Prueba de escritorio método obtenerProducto:

Nombre método	Parámetros	Resultados
obtenerProducto	double x, double y	Corresponde al resultado del producto de los dos números
	Supogan que x = 7; y = 4 x y 7 4	28

El programa para hallar la multiplicación de dos números e imprimir el resultado está representado en el diagrama de clases, que modela el producto de dos valores operados, y tiene implementados los métodos obtenerProducto() y main().

Producto
+obtenerProducto(x: double, y: double): double +main(args: String)

Y que codificado en Java es el siguiente:

```
public class Producto { /*
        * Método que retorna el producto de dos números
        */
public static double obtenerProducto(double x, double y) {
return x * y;
```

```
}

/**
        * Método principal de la clase
        */
        public static void main(String[ ] args) {
                double producto=0.0;
                /*
* Definición del objeto unaMultiplicacion de la clase
* Producto
                */
                Producto unaMultiplicacion = new Producto( );
                /*
                * el objeto unaMultiplicacion llama al método
 *obtenerProducto
                */
                producto=unaMultiplicacion.obtenerProducto(7, 8);
                System.out.println("El producto de 7 y 8 es "+producto);
        }
```

El método main() es el punto de entrada de un programa en Java, siendo el primer método que se ejecuta cuando se invoca a la máquina virtual de Java (JVM).

En el método main() se define el objeto unaMultiplicacion de la clase Producto, objeto que invoca al método obtenerProducto, el cual retorna el resultado, que es almacenado en la variable producto para ser impreso en pantalla.

Como se puede observar, en el nombre de la clase (Producto) la primera letra está en mayúscula, mientras que los métodos (obtenerProducto) y el objeto (unaMultiplicacion) inician con letras minúsculas, pero como tienen más de una palabra, las demás inician con letra mayúscula.

Además es recomendable inicializar el valor de las variables, por ejemplo, producto = 0.0, porque es un dato de tipo double; si fuera int, se haría con un cero (producto = 0;); un String se inicializa con comillas dobles (" "): String nombre = " "

4.9 Métodos que no retornan valor

Depende del requerimiento o lógica con que se desarrolle el ejercicio. Cuando no es necesario algún valor devuelto, se declara con la instrucción void.

```
void nombre método() {
   bloque o conjunto de acciones
}
```

4.10 Requerimientos

- Son una descripción de las necesidades de un usuario para resolver un problema o alcanzar un objetivo.
- Los requerimientos reflejan las necesidades de los clientes de un sistema que ayude a resolver algún problema.
- Los requerimientos para un sistema de software determinan con precisión lo que hará el sistema y definen las restricciones de su operación e implementación, para satisfacer las necesidades planteadas, siendo una descripción detallada del software, la cual puede servir como una base para diseño o implementación.

Los requerimientos pueden ser:

- Funcionales
- No funcionales

Para la redacción de los requerimientos, se debe tener en cuenta:

- Emplear el siguiente formato para expresar los requerimientos (formato Listado de requerimientos):

Requerimiento	Ejemplo
R1 descripción	Ejemplo del R1
R2 descripción	Ejemplo del R2

- Utilizar un lenguaje de una manera consistente, que permita diferenciar claramente entre requerimientos, resaltando los aspectos importantes del requerimiento.
- Evitar en lo posible los términos técnicos.

Para la documentación de cada requerimiento, se recomienda un formato como el siguiente:

Nombre	R1 nombre requerimiento 1
Descripción	Descripción puntal del requerimiento
Información de entradas	Valores de entrada
Información de salidas	Resultados o valores de salida

4.11 Un desarrollo completo

El banco El Ahorrador requiere construir un programa que permita simular la apertura de las cuentas desde $0 o con una determinada cantidad de dinero que el usuario disponga y además poder realizar las siguientes transacciones:

- Depositar dinero.
- Hacer retiros.
- Obtener el saldo actual de la cuenta.

A partir del anterior enunciado, determinar los requerimientos funcionales: Listado de los requerimientos del contexto planteado.

Requerimiento	Ejemplo
R1 Apertura de cuentas con $0	saldo = $0
R2 Apertura de cuentas con una determinada cantidad de dinero	saldo = $50.000 saldo = $1.450.000
R3 Consignar	Consignar una determinada cantidad y actualizar el valor del saldo.
R4 Retirar	Retirar una cuantía determinada de dinero y actualizar el saldo.
R5 Conocer el saldo actual	Saldo = $15.000

Documentación de cada requerimiento:

Nombre	R1 apertura de cuenta con $0
Descripción	Realizar la apertura de la cuenta con un valor de $0 es una operación que se realiza una sola vez.
Entradas	El valor de cero.
Salidas	El saldo de la cuenta se inicializa con un valor de $0.

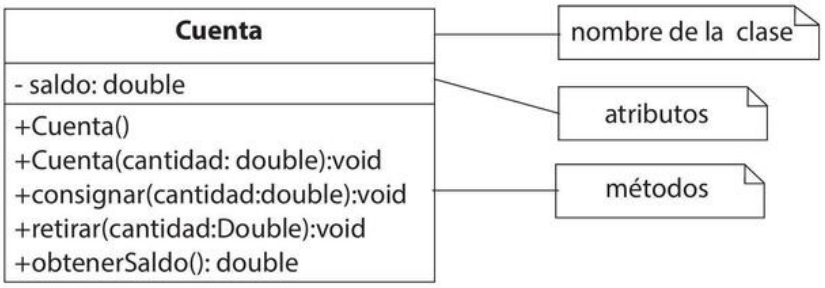
Nombre	R2 apertura de cuentas con una determinada cantidad de dinero
Descripción	Se abre una cuenta con el valor determinado.
Entradas	Cantidad de dinero, determinada por el cliente para abrir la cuenta.
Salidas	El saldo de la cuenta se inicializa con el valor de la cuantía que el usuario determina.

Nombre	R3 Consignar
Descripción	Valor que el cliente deposita en el banco e incrementa el valor del saldo.
Entradas	Valor consignado por el cliente.
Salidas	El valor del saldo incrementado con el valor de la consignación.

Nombre	R4 Retirar
Descripción	Valor que el cliente retira del banco disminuyendo el valor del saldo.
Entradas	Valor del retiro.
Salidas	El nuevo valor del saldo disminuido con la cuantía de dinero retirada.

Nombre	R5 Conocer saldo actual
Descripción	Valor del saldo actual
Entradas	Solicitud del saldo.
Salidas	Impresión en pantalla del valor del saldo actual.

El siguiente gráfico corresponde al diagrama de clases, el cual identifica y asocia las características de la clase (atributos) y las operaciones que se van a realizar (métodos).



Atributos:

Corresponde al saldo y es un dato de tipo double.

Nombre de la clase	Nombre del atributo	Nombre de la codificación	Objetivo
Cuenta	saldo	saldo	Atributo que permite conocer el estado actual del saldo de la cuenta y que se actualiza según sea el tipo de transacción (consignación o retiro).

Métodos:

A continuación, se documentan los atributos, métodos de la clase Cuenta, con dos constructores:

- Cuenta(); método que permite la apertura de la cuenta con $0; correspondiente por lo general a cuentas de nómina.
- Cuenta(double cantidad); constructor que permite la apertura de la cuenta con un saldo o valor inicial; correspondiente a clientes del banco.
- consignar(double cantidad); método que aumenta el valor del saldo cada vez que se realiza una consignación o, en otras palabras, cuando es invocado el método.

- retirar(double cantidad); método que disminuye el valor del saldo según cantidad retirada.
- double obtenerSaldo( ); método que retorna el valor del saldo actual existente en la cuenta.

Documentación métodos**:**

Análisis

- Entrada corresponde al valor de la consignación.
- No hay retorno de datos, ya que el valor de la cantidad se suma al valor del saldo.

Nombre del método	consignar
Entrada (lista de parámetros)	double cantidad
Resultado (tipo de dato de retorno)	No hay retorno.
Solución planteada	El valor del saldo se aumenta según cantidad consignada.
Firma del método	consignar(**double** cantidad)

Implementación del método:

```
public void consignar(double cantidad) {
        saldo = saldo + cantidad;
}
```

Prueba de escritorio método consignar:

Nombre método	Parámetros	Resultados Aumento del saldo
consignar	double cantidad	Suponga que se tiene un saldo = $0
	Cantidad 100	Saldo 100
	Cantidad 250	Saldo 350

Método que permite realizar los retiros en la cuenta creada.

Documentación métodos:

Análisis

- Entrada corresponde al valor del retiro.
- No hay retorno de datos, ya que el valor de la cantidad se resta al valor del saldo.

Documentación del método retirar elaborado en la siguiente tabla.

Nombre del método	retirar
Entrada (lista de parámetros)	double cantidad
Resultado (tipo de dato de retorno)	Ninguno
Solución planteada	El valor del saldo disminuye según valor de la transacción
Firma del método	retirar(**double** cantidad)

Implementación del método:

```
public void retirar(double cantidad) {
        saldo = saldo - cantidad;
}
```

Prueba de escritorio método retirar:

Nombre método	Parámetros	Resultados Aumento del saldo
retirar	double cantidad	Suponga que se tiene un saldo = $350
	Cantidad 100	Saldo 250

Implementación de la clase Cuenta en el lenguaje de programación de Java:

```
/**
 * Clase que modela la apertura de la cuenta y los movimientos de
 * la consignación y retiro de la misma
 * @author Miguel Hernández
 * @version 1.0
 */
public class Cuenta {
        private double saldo;
        /**
         * Se crea una cuenta con un saldo de $0
         */

        public Cuenta( ) {
                this.saldo=0;
        }

        /**
         * Se crea una cuenta con un saldo de acuerdo al valor de la apertura
         */
        public Cuenta(double cantidad)          {
                this.saldo=cantidad;
```

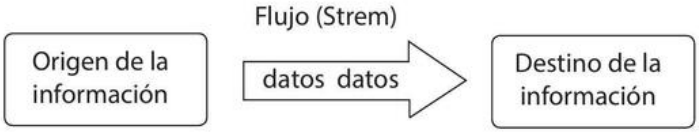
```
        }

        /**
         *  Se incrementa el valor del saldo según valor
    *de la consignación
         * @param cantidad
         */

        public void consignar(double cantidad) {
                saldo = saldo + cantidad;
        }
        /**
         * Se disminuye el valor del saldo según valor del retiro
         * @param cantidad
         */
        public void retirar(double cantidad) {
                saldo = saldo - cantidad;
        }
/**
         *
         * Método que retorna el valor del saldo de la cuenta
         * @return saldo
         */
        public double obtenerSaldo()
        {
                return saldo;
        }
}
```

4.12 Entrada y salida de datos

Las aplicaciones o programas desarrollados para las computadoras operan a partir de un conjunto de datos de entrada, con los cuales se realizan procesos y cálculos generando algún tipo de salida.

La forma de representar la entrada y salida de datos en Java es en base a streams (flujos de datos). Un stream es una conexión entre el programa y la fuente o destino de los datos. La información es procesada en serie, un carácter a continuación de otro, a través de esta conexión. Por ejemplo, cualquier dato que sea leído o escrito en un dispositivo de consola, de archivo, red o memoria

es un stream. Entre los datos que pueden leerse o escribirse se encuentran: los caracteres, los dígitos, los objetos de Java, los sonidos, las imágenes, entre otros.



4.13 Entrada y salida estándar

En Java, la entrada por teclado y la salida en pantalla se establece a partir de la clase System. Esta clase agrupa y define métodos y objetos que tienen relación con el sistema local para la lectura e impresión de datos.

System.in: objeto de la clase InputStream definido para tomar datos desde la entrada estándar del sistema, que habitualmente es el teclado.

System.out: objeto de la clase PrintStream que imprimirá los datos en la salida estándar del sistema, correspondiente a la pantalla donde el atributo out de la clase System tiene implementado el método println que integra el conocido System.out.println.

Para leer datos desde teclado, se utilizan clases que se encuentran disponibles en Java, dentro de las que están:

La clase Scanner:

Permite definir objetos para leer información desde una fuente de datos, que puede ser un archivo, una cadena de caracteres o el teclado, entre otros.

Para hacer uso de los métodos que implementa la clase Scanner, es necesario declarar una variable (objeto); este tipo de dato se asocia con el teclado (System.in).

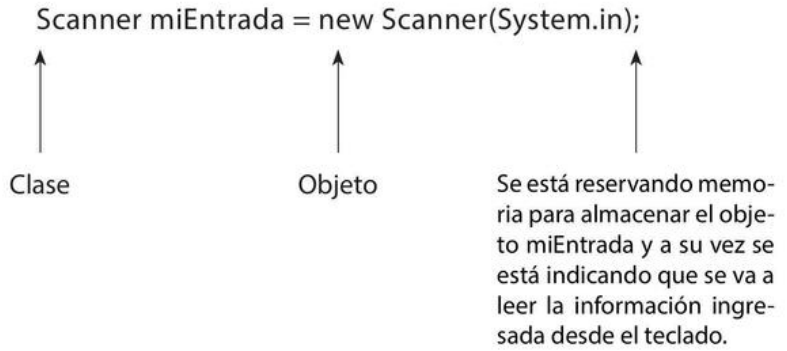
La sintaxis para definir un objeto de la clase Scanner es la siguiente:

Scanner nombre del objeto = new Scanner(System.in);

Por ejemplo:

- Scanner miEntrada = new Scanner(System.in);
- Scanner miTeclado = new Scanner(System.in);
- Scanner input = new Scanner(System.in);
- Scanner in = new Scanner(System.in);

Creación de un objeto de la clase Scanner:



Una vez definido el objeto, se puede invocar cualquiera de los métodos de la clase Scanner, dentro de los que están:

- nextInt( ): obtiene un número entero.
- nextDouble( ): obtiene un número real.
- nextLine( ): obtiene una cadena de caracteres.
- next( ): obtiene una cadena de caracteres hasta encontrar un espacio.
- nextShort( ): para leer datos de tipo short.
- nextFloat( ): para leer datos de tipo float.
- nextLong( ): para leer datos de tipo long.
- nextByte(): para leer datos de tipo byte.

No existen métodos para obtener datos de tipo boolean ni para datos de tipo char; para este último caso, se puede utilizar nextLine().charAt(0).

Ejemplo:

Programa que calcula el producto de dos números enteros o reales (números que pueden o no tener fracción decimal), donde los valores son ingresados por teclado.

Siguiendo el mismo modelo del programa anterior, lo que se debe realizar es:

- Para la lectura de los datos, se define un objeto de la clase Scanner.
- Scanner input = new Scanner(System.in).
- Se definen las variables, double multiplicando y double multiplicador.
- Se realiza la lectura de los datos, multiplicando=input.nextDouble( ).

Para utilizar la clase Scanner, se debe digitar al inicio del programa: **import** java.util.Scanner;, que significa importar la clase del paquete Java útil.

Ejemplo:

El siguiente código de Java corresponde a una clase que modela operaciones básicas de suma, resta, producto y división.

```
/**
 * Clase que modela las operaciones aritméticas básicas
 * @author (Miguel Hernández Bejarano)
 * @version 1.0
 */
public class Calculadora {
  private double x;
  private double y;

  public Calculadora() {
    this.x = 0;
    this.y = 0;
  }

  public Calculadora(double x, double y) {
    this.x = x;
    this.y = y;
  }

  public double getX() {
    return x;
  }

  public void setX(double x) {
```

```
    this.x = x;
  }

  public double getY() {
    return y;
  }

  public void setY(double y) {
    this.y = y;
  }

  public double obtenerSuma(){
    return x+y;
  }
  public double obtenerResta(){
    return x-y;
  }

  public double obtenerProducto(){
    return x*y;
  }

   public double obtenerDivision(){
    return x/y;
  }
}
```

La sentencia return se utilizar para finalizar el método y retornar el resultado del correspondiente proceso.

```
import java.util.Scanner;

public class PruebaCalculadora {

        //se define el objeto cassio
        private  Calculadora cassio;

        public PruebaCalculadora() {
                //se instancia el objeto cassio de la clase operación
                this.cassio=new Calculadora();
```

```
        }


        public void obtenerOperaciones(){
                double a;
                double b;
                Scanner lector = new Scanner(System.in);
                double resultado;
                //se leen los datos por teclado
                System.out.print("Digite Numero : ");
                a= lector.nextDouble();
                System.out.print("Digite Numero : ");
                b= lector.nextDouble();
                //actualizacion del estado del objeto cassio
                cassio.setX(a);
                cassio.setY(b);

                resultado=cassio.obtenerSuma();
                System.out.println("Resultado Suma : "+resultado);
                resultado=cassio.obtenerResta();
                System.out.println("Resultado Resta : "+resultado);
                resultado=cassio.obtenerProducto();
                System.out.println("Resultado Producto : "+resultado);
                resultado=cassio.obtenerDivision();
                System.out.println("Resultado División : "+resultado);

        }



        public static void main(String[] args) {

                PruebaCalculadora unaCalculadora=new PruebaCalculadora();
                unaCalculadora.obtenerOperaciones();
        }
}
```

La clase BufferedReader

Es una clase derivada que permite la lectura o introducción de datos por medio del teclado. Para utilizar esta clase, se deben importar los paquetes:

- BufferedReader
- InputStreamReader
- IOException

Definición de un objeto de la clase BufferedReader:

BufferedReader br = new BufferedReader (new InputStreamReader (System. in));

- br es el nombre del objeto.
- La clase InputStreamReader establece el paso del flujo de bytes a flujos de caracteres.
- El uso del BufferedReader requiere forzosamente del manejo de excepciones, por ello también es necesario importar la clase IOException (el tema de excepciones será abordado en un próximo curso).

Definición de un objeto
BufferedReader br = new BufferedReader (new InputStreamReader (System. in));

Ingreso de datos
Lectura de datos por teclado y almacenado en una variable.
String valor = br.readLine();

La lectura de datos por teclado corresponde a cadenas de texto, por lo tanto, el valor es de tipo String, por ello la asignación de la lectura se realiza de manera directa a la variable.

Cuando se requiere leer valores numéricos (enteros o con fracción decimal), se debe realizar una "conversión" de la cadena leída a número; por ejemplo, si nosotros introducimos por teclado 123, el programa lo tomará como texto y no como número, por lo que si queremos hacer operaciones con esos números, debemos convertir la lectura a entero con el método "parse" y almacenarla en una variable del mismo tipo, por ejemplo:

```
int dato;
dato = Integer.parseInt(br.readLine());
```

La estructura para realizar el proceso de entrada de datos por teclado es:

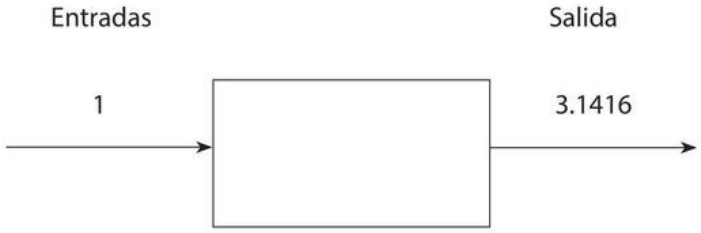
```
BufferedReader teclado = new BufferedReader(new InputStreamReader(System.in));
```

IOException

Ejemplo:

Programa que, a partir del radio, calcula el diámetro, longitud y área de la circunferencia. Con el objeto de facilitar su implementación y mantenimiento, las aplicaciones se dividen en módulos. Para este ejercicio, se diseñan cuatro métodos, tres para cumplir con los requerimientos de obtener el diámetro, obtener la longitud, obtener el área y el método main de la clase.

Método para calcular la longitud de la circunferencia a partir del radio:

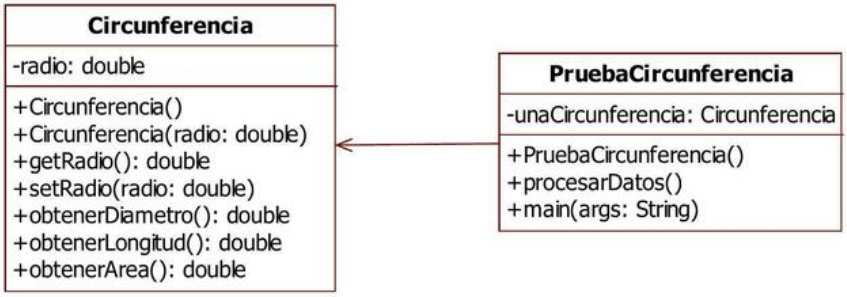


Nombre del método	obtenerLongitud
Entrada (lista de parámetros)	Ninguno
Resultado (tipo de dato de retorno)	double
Solución del problema	Se multiplica el valor del radio por 2; por la constante PI y se retorna el resultado
Firma del método	obtenerLongitud(int radio)

```
public double obtenerLongitud() {
                return 2 * Math.PI * radio;
}
```

- Math es una clase de Java.
- PI corresponde al valor de la constante de pi.

Diagrama de clases:




```
public class Circunferencia {
        private double radio;

        public Circunferencia() {
                this.radio = radio;
        }

        public Circunferencia(double radio) {
                this.radio = radio;
        }


        public double getRadio() {
                return radio;
        }

        public void setRadio(double radio) {
                this.radio = radio;
        }

        /**
         * método que calcula y devuelve el diámetro
```

```
        */
        public double obtenerDiametro() {
                return 2 * radio;
        }

        /**
        * método que calcula y retorna la longitud de la circunferencia
        */
        public double obtenerLongitud() {
                return 2* Math.PI * radio;
        }

        /**
        * método que calcula y retorna el área de la circunferencia
        */

        public double obtenerArea() {
                return Math.PI * Math.pow(radio,2) ;
        }

}


import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class PruebaCircunferencia {
        /**
        * Definción del objeto de la clase Circunferencia
        */
        private Circunferencia unaCircunferencia;

        public PruebaCircunferencia() {
                this.unaCircunferencia = new Circunferencia();
        }
        /**
        * Método que realiza la lectura e impresión de los
    *resultados
        * @throws IOException
        */
```

```
public void procesarDatos() throws IOException {
                double radio = 0;
                String numero = "";
                BufferedReader teclado = new BufferedReader(new
InputStreamReader(System.in));
                 System.out.println("Digite el valor del Radio:");
                numero = teclado.readLine();
                radio = Double.parseDouble(numero);
                 unaCircunferencia.setRadio(radio);
System.out.println("Valor Radio: "+ unaCircunferencia.getRadio()+ "\nDiame-
tro "+unaCircunferencia.obtenerDiametro()+
                "\nLongitud Circunferencia:
"+unaCircunferencia.obtenerLongitud()+
                "\nÁrea Circunferencia: "+ unaCircunferencia.obtenerArea());
        }

        public static void main(String[] args) throws IOException {
                PruebaCircunferencia prueba = new PruebaCircunferencia();
                prueba.procesarDatos();

        }
}
```

La clase Math

Representa la librería matemática de Java y tiene una serie de métodos definidos que permiten obtener el cálculo de las operaciones que se usan cotidianamente, como: raíz cuadrada, valor absoluto, logaritmo, potencias, funciones trigonométricas, entre otras.

Métodos:

- Math.abs(x) para int, long, float y double.
- Math.sin(double) Calcula el seno de un dato tipo double.
- Math.cos(double) Calcula el coseno de un double.
- Math.tan(double). Calcula la tangente.
- Math.pow(a,b). Eleva el número a a la potencia b.
- Math.round(x) para double y float.
- Math.random(). Devuelve un double aleatorio.
- Math.log(double). Devuelve el logaritmo de base e del valor.
- Math.sqrt(double). Devuelve el redondo de la raíz cuadrada del valor.

Constantes:

Math.E para la base exponencial.
Math.PI para PI.

Ejemplo:
Método que a partir de la base y el exponente retorna el resultado de la potencia.

```
public double obtenerLongitud(int base, int exponente) {
        return Math.PI * pow(base,expo) ;
}
```

Suponga que la base es 2 y el exponente 5, el valor retornado es 32.

4.14 Lecturas recomendadas

- Métodos.
- Parámetros.
- Métodos de la clase.
- Método del objeto.
- Sentencia return.
- Sentencia void.
- Clase String.
- Sobrecarga de constructores.
- Sentencias this y return.
- Encapsulamiento.

4.15 Preguntas de revisión de conceptos

- ¿Qué diferencia hay entre los métodos set y un constructor?
- ¿Cuál es la importancia de implementar los métodos set y get?
- ¿Qué es un constructor por defecto?
- Importancia de los métodos.
- ¿Cuántas entradas puede tener un método?
- ¿Cuántos valores puede retornar un método?
- ¿Cómo realizar la lectura de un dato de tipo char?

4.16 Ejercicios

1. Elaborar el diagrama de clases y la codificación del constructor, los métodos set y get, para cada uno de los siguientes enunciados:

   - Se requiere modelar la información de un libro conformada por el título, editorial, número de páginas y el autor.
   - Modelar la información correspondiente al teléfono, conformada por la referencia, costo, marca, modelo.

2. Construir la clase Docente, la cual estará integrada por los siguientes atributos:

Visibilidad	Nombre atributo	Tipo
private	nombre	String
private	documento	long
private	teléfono	String
private	e-mail	String

   - Crear uno o más constructores para la clase.
   - Crear los métodos get y set para los distintos atributos de la clase.
   - Elaborar una clase que implemente los métodos para obtener la raíz cuadrada, el logaritmo en base 10 y el logaritmo en base e.
   - Diseñar una clase que provea los métodos para las funciones trigonométricas de seno, coseno y tangente, expresados en radianes.

4.17 Bibliografía

Aguilar, L. J. (2004). *Fundamentos de programación algoritmos y estructura de datos*. Tercera edición. Ciudad: México. McGraw-Hill.

Booch, G. (1996). *Análisis y diseño orientado a objetos con aplicaciones*. Ciudad: México. Addison Wesley.

Deitel y Deitel (2012). *Cómo programar en Java*. Ciudad: México. Editorial Pearson (Prentice Hall).

Villalobos, J. y Casallas, R. (2006). *Fundamentos de programación, aprendizaje activo basado en casos*. Ciudad: México. Editorial Pearson.

Capítulo 5

Condicionales

5.1 Temática a desarrollar

- Estructuras de control
- Condicionales
- Operadores relacionales
- Operadores lógicos
- if-else
- switch-case
- Operador condicional (?)

5.2 Introducción

Las estructuras de control condicionales permiten realizar la comparación de un atributo o variable con otro u otros valores y, con base en el resultado de esa evaluación, se siguen determinadas acciones dentro del programa.

Las instrucciones que definen los condicionales en Java son: el if-else, el switch-case y el operador condicional (? o alternativa).

La sentencia if evalúa la expresión de la condición y, en caso de ser verdadera, ejecuta las instrucciones que están dentro de su bloque de if.

La estructura de selección if-else permite especificar una acción a realizar cuando la condición es verdadera y otra distinta cuando la condición es falsa, es decir, ejecuta el bloque if si la condición se cumple y, en caso contrario, se ejecuta el bloque de instrucciones del else.

El operador condicional (?) evalúa una expresión booleana y determina cuál de las acciones se debe ejecutar en base al resultado de la comparación.

La figura 19 amplía el concepto de las condicionales en Java, incluyendo el uso de operadores:

En el desarrollo del capítulo y a través de ejemplos concretos, se utilizan buenas prácticas de programación que van desde el modelado de la situación planteada a la solución en Java, incluyendo estilos de programación dentro del código fuente, como la identación, el uso de los constructores, la identificación de los métodos, la declaración y nombre de las variables locales, el uso obligatorio de los corchetes, los espacios después de una palabra clave y antes del corchete, entre otros aspectos.

5.3 Estructuras de control condicionales

Son elementos del lenguaje de programación que permiten tomar decisiones en un programa; las estructuras de selección permiten tomar distintos cursos de acción de acuerdo a una condición, que viene dada por una expresión lógica. Las condiciones hacen uso de los operadores relacionales y/u operadores lógicos.

Ejemplos:

- (X >= Y).
- (X >= Y) && (X>= Z).
- (A != 0) || (A != 5).

5.4 Operadores relacionales

Son símbolos que se usan para establecer una relación para evaluar dos o más expresiones y/o valores (combinación de operadores y variables) donde el resultado de la comparación puede ser verdadero o falso.

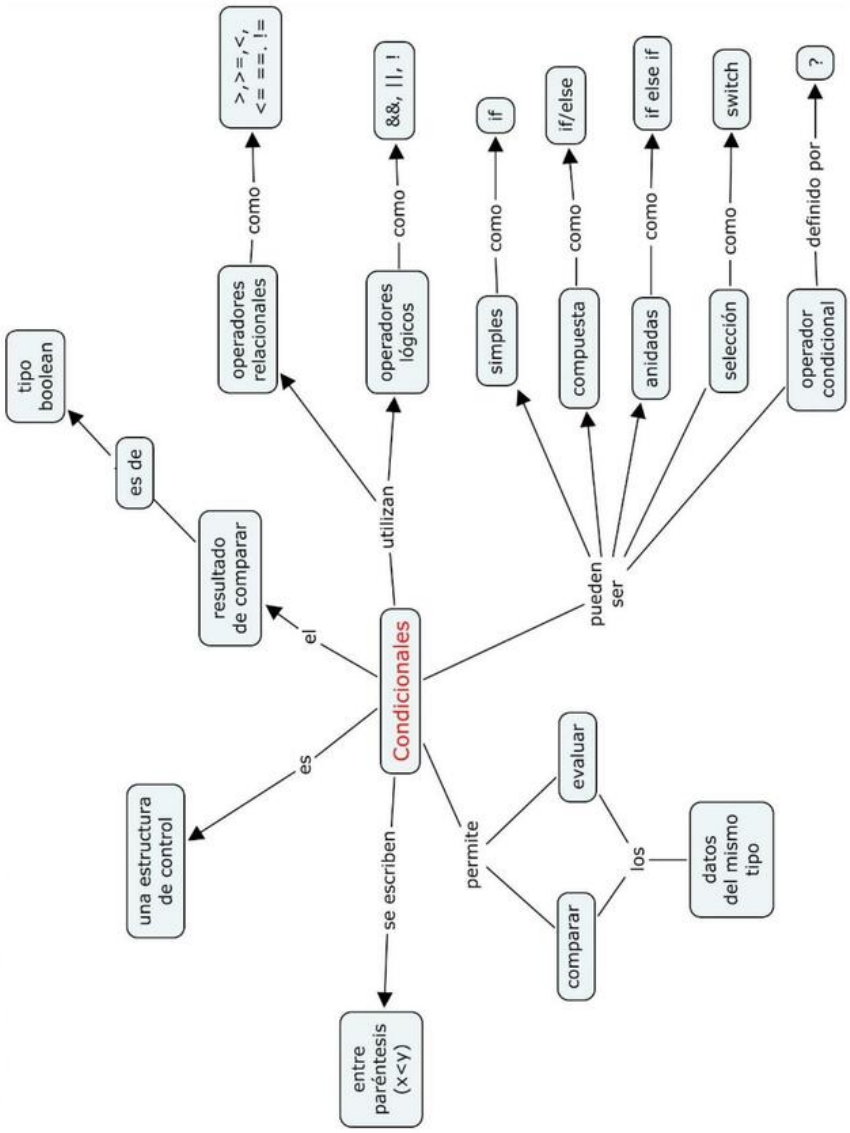


Figura 19.
Concepto de las estructuras condicionales

La tabla 7 presenta los operadores relacionales utilizados en Java:

Tabla 7.
Operadores relacionales

Operador	Nombre	Utilización	Resultado
>	Mayor	X > Y	Verdadero si X es mayor que Y
>=	Mayor e igual	X >= Y	Verdadero si X es mayor o igual que Y
<	Menor	X < Y	Verdadero si X es menor que Y
<=	Menor e igual	X <= Y	Verdadero si X es menor o igual que Y
==	Igual	X == Y	Verdadero si X es igual a Y
!=	Diferente	X != Y	Verdadero si X es distinto de Y

- 7 > 2 /* true */
- 3 >= 5 /* false */
- 14 = = 14 /* true */
- 32 != 32 /* false */

5.5 Operadores lógicos

Permiten construir expresiones lógicas. Java tiene los operadores lógicos definidos Y, O, NOT; los argumentos y resultados de estos operadores son de tipo booleano. En la tabla 8 se detallan estos operadores y se complementa con un ejemplo.

Estos se conocen con el nombre de conjunción, disyunción y negación. En la tabla 9 se presentan los resultados de aplicar los operadores lógicos para los posibles valores de entrada.

Tabla 8.
Operadores lógicos

Operador	Nombre	Descripción	Ejemplo	Retorna
&&	AND (y)	Conjunción	c > 7 && c <= 10	Retorna true si ambos operandos son verdaderos (true).
||	OR (o)	Disyunción	x == 1 || y == 2	Retorna true si alguno de los operandos es verdadero (true).
!	NOT (no)	Negación	! x	Niega el operando que se le pasa.

Tabla 9.
Resultado operadores lógicos

X	Y	X && Y	X || Y	!X
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

Ejemplo:

- (true || false) && true // true
- (false || false) && (!true) //false
- !true && !false //false
- !(4 > 8) || (9 > 0) && (4 == 2) //false

Los operadores relacionales y lógicos son fundamentales para implementar las condiciones de las sentencias de control.

5.6 Tipos de condicionales

5.6.1 Condicionales simples

Las sentencias condicionales permiten comparar el valor de una variable con otra, con valores numéricos, y con base en el resultado de esta comparación, se sigue un curso de acción dentro del programa. La sentencia que permite llevar a cabo la ejecución condicional es la palabra reservada if.

Si al evaluar la expresión el resultado es verdadero, entonces se ejecuta(n) las sentencias u operación(es) del bloque del if.

La notación para esta sentencia es la siguiente:

```
if ( Expresion boolean ) {
  instrucciones;
}
```

Por ejemplo:

- If (x > y): se compara si x es mayor que y.
- If (a == b): se evalúa si la variable a es igual a la variable b.
- If (x != VALOR_MAXIMO): se evalúa si x es diferente al valor de la constante.

Evaluar intervalos

La expresión del condicional puede utilizar los operadores relacionales y lógicos.

1. Para evaluar si un valor se encuentra entre un intervalo cerrado y el otro abierto (1.0,5.0], la condición sería:
   if (a>1.0 && a<=5.0)

   - Al evaluar si el intervalo no se encuentre entre 6 y 10:
   if (a<=6.0 || a>10)

- Negación (!) del valor de un intervalo:

```
if (!(a>1 && a<=10))
```

Suponga que a vale 8; la expresión interna de comparar es verdadera ya que 8 es mayor que 1 y menor que 10; al negar esta expresión, queda falsa.

Los condicionales le permiten validar datos.

Ejemplos:

1. En la operación de la división, el divisor no puede ser cero, se evalúa si este es igual a cero, para lo cual se imprime el mensaje: "error dividir por cero"; en caso contrario, se ejecuta la sentencia cociente = dividendo / divisor;.

```
if ( divisor == 0  ) {
    System.out.println("error dividir por cero");
} else {
        cociente = dividendo / divisor;
}
```

2. Otra forma de realizar el anterior proceso se da al cambiar el operador de la condición: en vez de igual, se utiliza el operador diferente (divisor ! = 0); al ser verdadera la condición, se ejecuta la sentencia cociente = dividendo / divisor;; en caso contrario, se imprime el mensaje "error dividir por cero".

```
if ( divisor != 0  ) {
        cociente = dividendo / divisor;
} else {
    System.out.println("error dividir por cero");
}
```

3. Para calcular la raíz cuadrada de un número, este no puede ser menor que cero, se evalúa si el número es menor que cero; en la sentencia if, al ser verdadero, se imprime el mensaje raíz imaginaria, en caso contrario, se ejecuta la sentencia raiz = Math.sqrt(*numero*);.

```
if ( numero < 0  ) {
    System.out.println("raíz imaginaria");
} else {
        raiz = Math.sqrt( numero );
```

```
}
```

Si la variable numero es mayor e igual que cero, se ejecuta la sentencia raiz = Math.sqrt(numero);, en caso contrario, se imprime el mensaje "raíz imaginaria".

```
if ( numero >= 0) {
        raiz = Math.sqrt( numero );
} else {
    System.out.println("raíz imaginaria");
}
```

4. Programa que permite calcular el descuento del 5 % para compras superiores a $100.000.

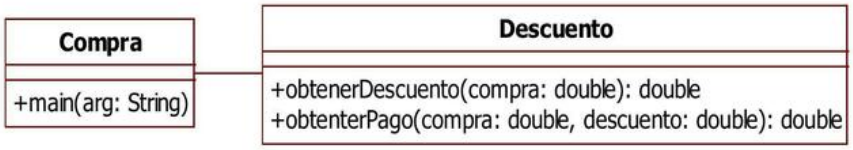
La aplicación se modela a partir de la clase Compra, que tiene como responsabilidad la lectura del valor de la compra e impresión de los valores de la compra, el descuento y el total a pagar.

La clase Descuento permite modelar los cálculos del descuento y el total a pagar.

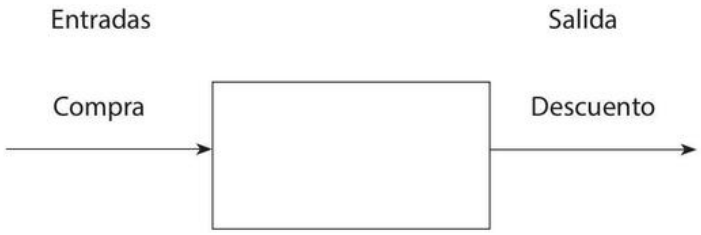
En Java, se recomienda definir constantes en vez de utilizar valores dentro de los programas, como, por ejemplo.

```
final double VALOR_COMPRA_DESCUENTO = 100000;
final double VALOR_DESCUENTO = 0.05;
```

Diagrama de clases:



La entrada de datos del **método se da a partir de la** compra y la salida es el descuento que depende del monto de la compra.



Si la compra es menor que 100.000, no tiene descuento, en caso contrario, se calcula el descuento. Como apoyo para la implementación del método, se utiliza la tabla 10.

Tabla 10.
Apoyo implementación del método

Nombre del método	obtenerDescuento
Entrada (lista de parámetros)	Ninguno
Resultado (tipo de dato de retorno)	double valorCompra
Solución del problema	Se evalúa si la compra es mayor o igual que 100.000, para calcular el descuento
Firma del método	obtenerDescuento ( )

De la anterior tabla se puede construir la cabecera del método:

Tipo retorno	Nombre del método	Lista parámetros
double	obtenerDescuento	**( )**

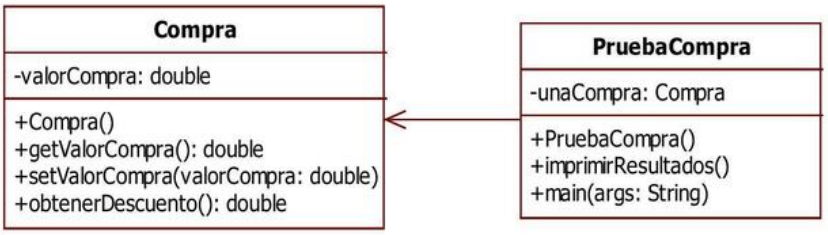
```
if ( compra >= VALOR_COMPRA_DESCUENTO) {
        descuento = compra * VALOR_DESCUENTO;
}
```

Las variables se inicializan cuando se definen:
```
double descuento = 0.0;
```

La comparación compra >= VALOR_COMPRA_DESCUENTO es una expresión booleana.

La sentencia descuento = compra * VALOR_DESCUENTO; se ejecuta solamente si la expresión booleana es verdadera.

Diagrama de clases:



Codificación en Java del modelado del problema a partir del diagrama de clases:

```
public class Compra {
        private double valorCompra;

        public Compra() {
                this.valorCompra = 0;
        }

        public double getValorCompra() {
                return valorCompra;
        }

        public void setValorCompra(double valorCompra) {
                this.valorCompra = valorCompra;
        }

        public double obtenerDescuento() {
                double descuento = 0.0;
                final double VALOR_COMPRA_DESCUENTO = 100000;
                final double VALOR_DESCUENTO = 0.05;
                if ( valorCompra >= VALOR_COMPRA_DESCUENTO) {
                        descuento = valorCompra * VALOR_DESCUENTO;
```

```
                }
                return descuento;
        }
}
```

Clase que modela la prueba de la compra

```
public class Compra {
        private double valorCompra;

        public Compra() {
                        this.valorCompra = 0;
        }

        public double getValorCompra() {
                return valorCompra;
        }

        public void setValorCompra(double valorCompra) {
                this.valorCompra = valorCompra;
        }

        public double obtenerDescuento() {
                double descuento = 0.0;
                final double VALOR_COMPRA_DESCUENTO = 100000;
                final double VALOR_DESCUENTO = 0.05;
                if ( valorCompra >= VALOR_COMPRA_DESCUENTO) {
                        descuento = valorCompra * VALOR_DESCUENTO;
                }
                return descuento;
        }
}
```

5.6.2 Condicionales compuestas

Están conformadas por un if y un else que permiten optar entre dos opciones o alternativas posibles en función del cumplimiento o no de una determinada condición.

La sintaxis de una sentencia condicional if else es la siguiente:

```
if (expresion_booleana) {
        sentencia_de_verdadero;
}
else {
        sentencia_de_falso;
}
```

1. Aplicación que permite determinar, a partir de la edad de una persona, si es mayor o menor de edad.

Se construyen dos clases: en la clase Persona se implementa el método que evalúa si es mayor o menor de edad y en la clase PruebaPersona se lee la edad de la persona.

edad = lea.nextInt();

Diagrama de clases:

```
if ( edad >= 18 ) {
        resultado = "Es una persona mayor de edad";
} else {
        resultado = "Es una persona menor de edad";
}
```

- Si la condición (edad >= 18) es verdadera, se ejecuta la sentencia resultado = "Es una persona mayor de edad";, en caso contrario, se ejecuta la sentencia resultado = "Es una persona menor de edad";.
- Ambas sentencias (if o else) no se ejecutarán al mismo tiempo, esto quiere decir que son excluyentes.

Codificación en Java del modelado del problema a partir del diagrama de clases:

```
/**
* Clase que modela la edad de la persona
*/
```

```
public class Persona {
        private int edad;
        public Persona() {
                this.edad = 0;
        }

        public Persona(int edad) {
                if (edad > 0){
                this.edad = edad;
          } else {
                this.edad = 0;
          }
        }

        public int getEdad() {
                return edad;
        }

        public void setEdad(int edad) {
                this.edad = edad;
        }

        public String evaluarEdad () {
                String resultado = "";
                //Este es un condicional que compara números
                if ( edad >= 18 ) {
                        resultado = "Es una persona mayor de edad";
                }else{
                        resultado = "Es una persona menor de edad";
                }
                return resultado;
        }
}


import java.util.Scanner;

public class PruebaPersona {
        public static void main (String arg[]) {
                Scanner lea = new Scanner(System.in);
                Persona unaPersona = new Persona();
```

```
                int edad = 0;
                System.out.print("Digite edad de la persona ");
                edad = lea.nextInt();
                unaPersona.setEdad(edad);
                System.out.println(unaPersona.evaluarEdad()+" tiene
"+unaPersona.getEdad()+" años");
        }
}
```

5.6.3 Condicionales anidadas

Consisten en que, dentro de una sentencia if, se puede colocar otra sentencia if; en ocasiones se requiere anexar un if a continuación de otro if, o un if-else. En otras palabras, un condicional anidado es un bloque condicional dentro de otro, como lo señala la siguiente notación:

```
if (condición1) {
                instrucción1;
        } if (condición2) {
                instrucción2;
        } if (condición3) {
                instrucción3;
        }
        ...
        else {
                instrucciones del else
        }
}
```

- Dentro del bloque de sentencias del if (o de los bloques de if-else) puede encontrarse otra sentencia if.
- Las llaves ({ }) son necesarias para agrupar las sentencias múltiples.
- Es conveniente minimizar el número de anidaciones.
- Se debe revisar el código de las condiciones para evitar errores.

La siguiente notación muestra la forma general de su empleabilidad:

```
if (condición1) {
                instrucción1;
        } else if (condición2) {
                Instrucción2;
```

```
        } else if (condición3) {
                Instrucción3;
        }
        ...
        else {
                instrucciones del else
        }
}
```

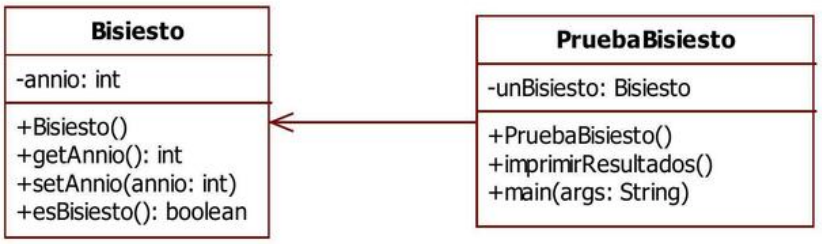
Ejemplos:

1. Programa que determina si un año es o no bisiesto.

   Aplicación que se diseña con la clase Bisiesto, que implementa el método para evaluar el año, y la clase PruebaBisiesto, que tiene como responsabilidad la lectura del año e impresión del si es o no un año bisiesto; en esta clase se define un objeto unBisiesto de la clase Bisiesto: Bisiesto unBisiesto = **new** Bisiesto();. El objeto unBisiesto invoca al método esBisiesto (unBisiesto.esBisiesto( ));, instrucción incluida dentro del condicional (**if** (unBisiesto. esBisiesto( ))), ya que el método retorna un valor booleano de true o false.

   Son bisiestos los años múltiplos de 4, los que son múltiplos de 100 y los que sean múltiplos de 400.

Diagrama de clases:



Codificación en Java del modelado del problema si un año es bisiesto a partir del diagrama de clases:

```
public class Bisiesto {
	private int annio;

	public Bisiesto() {
		this.annio = 2015;
	}

	public int getAnnio() {
		return annio;
	}

	public void setAnnio(int annio) {
		this.annio = annio;
	}

	public boolean esBisiesto( ) {
		boolean bisiesto = false;
		if (annio % 4 == 0) {
			if ((annio % 100 != 0) || (annio % 400 == 0)) {
				bisiesto=true;
			} else {
				bisiesto=false;
			}
		} else {
			bisiesto=false;
		}
		return bisiesto;
	}
}
```

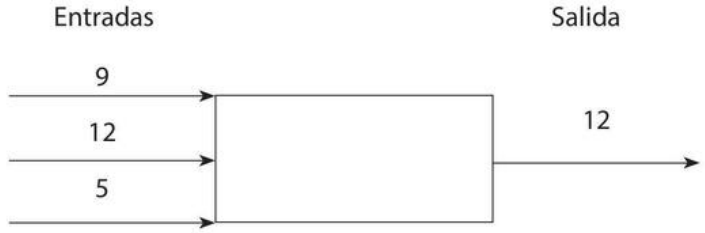
La clase PruebaBisiesto queda así:

```
import java.util.Scanner;

public class PruebaBisiesto {
	private Bisiesto unBisiesto;

	public PruebaBisiesto() {
```

```
            this.unBisiesto = new Bisiesto();
        }

        public void imprimirResultados() {
            int aa = 0;
            System.out.print("Digite el año: ");
            Scanner lea = new Scanner(System.in);
            aa = lea.nextInt();
            unBisiesto.setAnnio(aa);
            if(unBisiesto.esBisiesto()){
                System.out.println(unBisiesto.getAnnio()+ " es un año
bisiesto ");
            } else {
                System.out.println(unBisiesto.getAnnio()+ " No es un
año bisiesto ");
            }
        }


        public static void main(String[] args) {
            PruebaBisiesto unaPrueba = new PruebaBisiesto();
            unaPrueba.imprimirResultados();
        }
}
```

2. Método que evalúa tres números enteros diferentes y retorna el mayor de estos.

Etapa de análisis: son tres datos de entrada y una salida; por ejemplo, ingresa el 9, 12, 5; la salida es 12, que corresponde al número mayor.



Para que el ejercicio sea más general y pueda ser aplicado a diferentes valores, se utilizan tres variables x, y, z de tipo entero que corresponden a las entradas de datos.

Como apoyo a la parte de análisis, se completa en la tabla 11.

Tabla 11.
Apoyo parte de análisis

Nombre del método	obtenerMayor
Entrada (lista de parámetros)	Ninguno
Resultado (tipo de dato de retorno)	Int
Planteamiento del problema	Comparar las variables para determinar el mayor: • Si (x > y) y (x > z), el mayor es x. • Si (y > x) y (y > z), el mayor es y. • Si (z > x) y (z > y), el mayor es z.

La firma del método sería:

Nombre del método	Lista parámetros
obtenerMayor	( )

La implementación del método sería:

```
public int obtnerMayor( ) {
        int mayor = 0;
        if ( x  >= y &&  x >= z )  {
                mayor = x;
        }  else if( y >= x && y >= z ) {
                mayor = y;
        }  if( z >= x && z >= y ) {
                mayor=z;
        }
        return mayor;
}
```

En la prueba de escritorio de este método, se realiza el seguimiento para determinar si el método es correcto o si no, para realizar los ajustes correspondientes.

Suponiendo que x = 12; y = 9; z = 5, la primera condición a evaluar es si x es mayor que y; como es verdadero, evalúa si x es mayor que z; esto lo realiza la sentencia **if** (x >= y && x>=z). Como x es el número mayor, entonces la variable mayor almacena el número 12, tal como se indica a continuación:

x	y	z	Mayor
12	9	5	12

Si x = 9; y = 12; z = 5, evalúa **if** (x >= y && x>=z). Si x es mayor que y, como es falso, no evalúa; si x es mayor que z, continúa con la siguiente comparación, **else if** (y >= x && y >= z); y es mayor que x; y es mayor que z, entonces la variable y tiene almacenado el número mayor que es 12.

x	y	z	Mayor
9	12	5	12

La tabla completa sería:

x	y	z	Mayor
12	9	5	12
9	12	5	12

```
/**Sean x, y, z variables enteras las cuales se van
*  comparando para ir devolviendo el número mayor en cada
*  caso de la instrucción condicional
*/
public int obtnerMayor( ) {
        if( x >= y )
                if( x >= z )
                  return x;
                else
                  return z;
        else if( y >= z )
                  return y;
        else
                  return z;
}
```

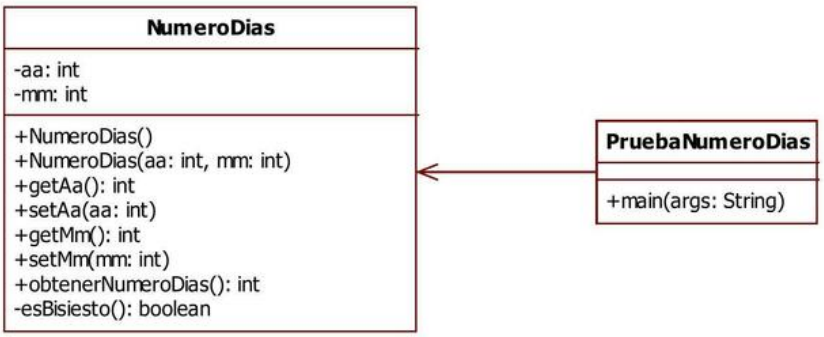
3. Programa que determina el número de días del mes teniendo en cuenta si el año es bisiesto, a partir del número del mes y del año.

Aplicación diseñada con dos clases: la clase NumeroDias, que tiene la responsabilidad de evaluar el número de días y determinar si el año es bisiesto, para lo cual se implementaron dos métodos, y la clase PruebaNumeroDias, que tiene la responsabilidad de leer el número del mes y año e imprimir el número de días.

- obtenerNumeroDias(), método que evalúa el número del mes; si este es 1, 3, 5, 7, 8, 10, 12, retorna 31; si no, compara si el mes es 2 invocando al método esBisiesto; en caso de ser verdadero, el retorno en número de días es 29 y 28 en caso contrario; para los demás meses 4, 6, 9, 11, retorna 30.

- El método esBisiesto(**int** aa) que tiene como parámetro el año, el cual lo evalúa en el condicional **if** ((aa % 4 == 0) && ((aa % 100 != 0) || (aa % 400 == 0))); en caso de ser verdadero, retorna true, de lo contrario, retorna false.

Diagrama de clases:



Codificación en Java del modelado del problema a partir del diagrama de clases:

```
/**
* Clase que modela el número de días del mes y
* evalúa si el año es bisiesto
*/
```

```
public class NumeroDias {
	private int aa;
	private int mm;

	public NumeroDias( ) {
		this.aa = 2015;
		this.mm = 4;
	}

	public NumeroDias(int aa, int mm) {
		this.aa = aa;
		this.mm = mm;
	}


	public int getAa() {
		return aa;
	}

	public void setAa(int aa) {
		this.aa = aa;
	}

	public int getMm() {
		return mm;
	}


	public void setMm(int mm) {
		this.mm = mm;
	}


	public int obtenerNumeroDias( ){
		int numeroDias = 0;
		if ((mm==1) || (mm==3) || (mm==5) || (mm==7) || (mm==8)
||  (mm==10) || (mm==12)) {
			numeroDias = 31;
		} else if (mm==2){
			if ( esBisiesto()) {
				numeroDias = 29;
```

```
                }else{
                        numeroDias = 28;
                }
        } else {
                numeroDias = 30;
        }
        return numeroDias;
    }


    /**
     * Método que evalúa si el año es bisiesto
     * @return true si es año bisiesto; false en caso contrario
     */

    private boolean esBisiesto(){
        boolean bisiesto = false;
        if ((aa % 4 == 0) && ((aa % 100 != 0) || (aa % 400 == 0))) {
                bisiesto=true;
        } else {
                bisiesto=false;
        }
        return bisiesto;
    }
}


/**
 * Clase que modela la lectura del año y mes e imprime el número de días del
mes
 */
import java.util.Scanner;

public class PruebaNumeroDias {
    public static void main(String[] args) {
        //definición objeto unBisiesto de la clase Bisiesto
        Scanner lea = new Scanner(System.in);
        NumeroDias numeroDias = new NumeroDias();
        int aa = 0;
        int mm = 0;
        System.out.print("Digite el año ");
```

```
                aa = lea.nextInt();
                System.out.print("Digite el mes ");
                mm = lea.nextInt();
                numeroDias.setAa(aa);
                numeroDias.setMm(mm);
        System.out.println("El Mes "+numeroDias.getMm()+", del año
"+numeroDias.getAa()+" es de " +numeroDias.obtenerNumeroDias()+" días ");
                System.exit(0);
        }
}
```

5.7 Selección múltiple

Cuando se requiere comparar una variable con una serie de datos diferentes, es recomendable utilizar la sentencia switch, en la cual se indican los posibles valores que puede tomar la variable y las sentencias que se tienen que ejecutar cuando el valor de la variable coincide con alguno de estos.

La estructura switch case permite resumir los if-else-if o if anidados.

La sintaxis de la instrucción switch case en Java es:

```
switch (expresión o variable de control)
{
   case exp 1:
      sentencia 1;
      sentencia 2;
      break;

   case exp 2:
   case exp N:
      sentencia N;
      break;
   default:
      sentencia D;
}
```

```
switch ( expresión ) {
        case valor1:
                break;
        case valor2:
                break;
        case valorN:
                break;
        default:
}
```

La instrucción switch toma el valor de la variable que se le pasa como argumento, el cual se compara con cada uno de los valores literales de las sentencias case, como se explica a continuación.

- La palabra reservada switch debe ir en el encabezado del bloque.
- La variable de control puede ser de tipo: byte, char, short o int.
- Los paréntesis que contienen la variable de control son obligatorios.
- La palabra reservada case ejecutará las sentencias correspondientes, con base en el valor de la variable de control, que deberá de evaluarse.
- La cláusula default es opcional; para el caso de que el dato de la variable no coincide con ningún valor, entonces se ejecutan las sentencias por default cuando las haya.
- La sentencia break al final de cada case pasa el control al final de la sentencia switch; de esta forma, cada vez que se ejecute un case, todos los enunciados case restantes son ignorados y termina la operación del switch.

El funcionamiento del switch

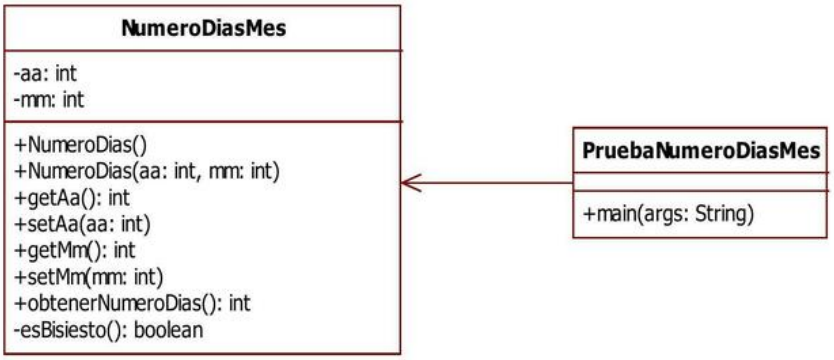
- Se compara la expresión o variable de control con el primer valor; si coincide, se ejecutan las instrucciones ubicadas bajo ese dato y todas las siguientes que se encuentren hasta hallar un break.
- Si no coincide, se compara con el segundo valor, y así sucesivamente.
- Al no coincidir con ningún valor, se ejecutan las instrucciones que haya en la parte default, si existe.
- Después de un break, la instrucción switch termina y continúa con la siguiente instrucción.

Programa que determina el número de días del mes, teniendo en cuenta si el año es bisiesto, haciendo uso de la sentencia switch.

Aplicación diseñada con dos clases: PruebaNumeroDiasMes, que tiene la responsabilidad de leer el número del mes y año e imprimir el número de días y la clase NumeroDiasMes, que tiene la responsabilidad de evaluar el número de días y determinar si el año es bisiesto, para lo cual se implementaron dos métodos:

- obtenerNumeroDias( )

- El método esBisiesto( )



- Si la expresión retorna la etiqueta 1, 3, 5, 7, 8, 10, 12, se ejecuta la sentencia numeroDias = 31;.
- Si la expresión retorna la etiqueta 2, invoca el método esBisiesto( );.
- Si la expresión retorna la etiqueta 4, 6, 9, 11, se ejecuta la sentencia numeroDias = 30;.
- Cuando ninguno de los casos corresponde, se ejecutará la sentencia del default System.*out*.println("Mes no contemplado").

Codificación en Java del modelado del problema a partir del diagrama de clases:

```
/**
 * Clase que modela el número de días del mes y
 * si un año es bisiesto
 *
 */
public class NumeroDiasMes {

        private int aa;
          private int mm;

          Implementación de los métodos set y get.

          public NumeroDiasMes(int aa, int mm) {
                    this.aa = aa;
                    this.mm = mm;
          }
```

```
public int obtenerNumeroDias(){
        int numeroDias = 0;
        switch(mes) {
        case 1: case 3: case 5: case 7: case 10: case 12:
        case 8:
                numeroDias = 31;
                break;
        case 2:
                if ( esBisiesto(año)) {
                        numeroDias = 29;
                }else{
                        numeroDias = 28;
                }
                break;
        case 4: case 6: case 9: case 11:
                numeroDias = 30;
                break;
        default:
                System.out.println("Mes no contemplado");
        }
        return numeroDias;
}



/**
 * Método que evalúa si el año es bisiesto
 * @return true si es año bisiesto; false en caso contrario
 */

private boolean esBisiesto(){
        boolean bisiesto = false;
        if ((aa % 4 == 0) && ((aa % 100 != 0) || (aa % 400 == 0))) {
                bisiesto=true;
        } else {
                bisiesto=false;
        }
        return bisiesto;
```

```
        }
  }


/**
 * Clase que modela la lectura del año y mes e impresión del número de días
del mes
 */
import java.util.Scanner;
public class PruebaNumeroDiasMes {
        public static void main(String[] args) {
                //definición objeto numeroDias de la clase NumeroDiasMes
                Scanner lea = new Scanner(System.in);
                NumeroDiasMes numeroDias = new NumeroDiasMes ();
                int aa = 0;
                int mm = 0;
                System.out.print("Digite el año ");
                aa = lea.nextInt();
                System.out.print("Digite el mes ");
                mm = lea.nextInt();
                numeroDias.setAa(aa);
                numeroDias.setMm(mm);
            System.out.println("El Mes "+numeroDias.getMm()+", del año
"+numeroDias.getAa()+" es de "
+numeroDias.obtenerNumeroDias()+" días ");
                System.exit(0);
        }

}
```

5.8 Operador condicional (?)

En Java, el operador (?), denominado alternativa, es un operador ternario, conformado por tres operandos, el cual permite bifurcar el flujo del programa a partir del valor de una variable boolean:

expresión1 ? expresión2 : expresión3; cuando el resultado de evaluar la expresión lógica es verdadero, retorna el valor de la segunda expresión y, en caso contrario, devuelve el valor de la tercera expresión.

resultado = (expresión lógica ? expresión2 : expresión3);

El operador condicional ? es similar a un if-else, por ejemplo:

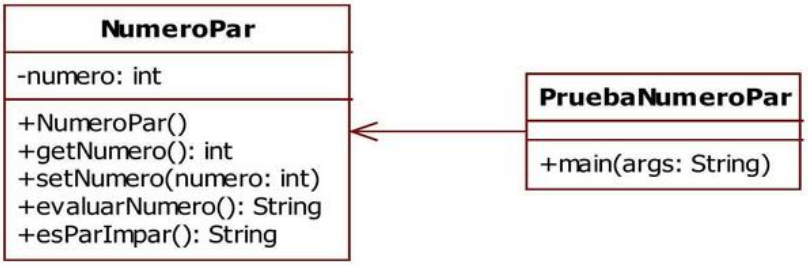
Si x es igual a cero, entonces a es igual a 1; en caso contrario, es igual a 2.	Si x es igual a cero, entonces a es igual a 1; en caso contrario, es igual a 2.

`if( x == 0 ) {`
`  a = 1;`
`} else {`
`  a = 2;`
`}`
	`a = (x == 0) ? 1 :2;`

Ejemplos:

1. Programa que evalúa si un número es par o impar utilizando el operador condicional (?).

Diagrama de clases:




```
/**
* Clase que modela un número evaluando si es par o impar
*
```

```
 */
public class NumeroPar {
        private int numero;

        public NumeroPar() {
                this.numero = 0;
        }

        public int getNumero() {
                return numero;
        }

        public void setNumero(int numero) {
                this.numero = numero;
        }

        /**
         * Método que evalúa si el número es par o impar
         * utilizando el operador alternativa
         * @return
         */

        public String evaluarNumero()
        {
                String respuesta=((numero % 2 == 0)?"par":"impar");
                return respuesta;
        }

        /**
         * Método que evalúa si el número es par o impar
         * utilizando las sentencias if-else
         * @return
         */

        public String esParImpar( )
        {
                String respuesta = "";
                if(numero % 2 == 0){
                        respuesta = "par";

                } else {
```

```
                        respuesta = "impar";
                }
                return respuesta;
        }
}


import java.util.Scanner;

/**
 * Clase que modela lectura e impresión de un número
 */

public class PruebaNumeroPar {
        public static void main(String[] args) {
                //definición de objetos
                Scanner lea = new Scanner(System.in);
                NumeroPar numero = new NumeroPar();
                //definición de variable
                int n = 0;

                System.out.print("Digite un número ");
                n = lea.nextInt();
                numero.setNumero(n);
            System.out.println("El número "+numero.getNumero()+", es
  "+numero.evaluarNumero());
        }

}
```

5.9 Preguntas de revisión de conceptos

- Importancia de los condicionales en Java.
- Semejanzas entre los condicionales anidados y la sentencia switch.
- Importancia de realizar pruebas de escritorio a los programas.
- ¿Qué ventaja tiene modelar una aplicación en diferentes clases?

5.10 Lecturas recomendadas

- Condicionales.
- Jerarquía de los operadores relacionales y lógicos.

5.11 Ejercicios

Construir las correspondientes sentencias de condicionales que satisfacen los siguientes requerimientos:

1. Hallar el mayor de dos números enteros diferentes.

_____

2. Comparar si una cantidad es superior a $150.000.

_____

3. Evaluar si un número entero positivo es múltiplo de tres.

_____

4. Determinar si un número es mayor que 20.

_____

5. A partir del siguiente método:

```
public void obtenerResultado( ) {
        int a = 15;
        int b = 25;
        int c = 40;
        boolean r=false;
        boolean s=false;
        boolean t=false;
        boolean u=false;

        r = a + b > c;
        s = a - b < c;
```

```
        t = a - b == c;
        u = a * b != c;
        System.out.println(r + " " + s + " " + t + " " + u);
}
```

Determinar (se sugiere realizar el seguimiento en papel prueba de escritorio) cuáles valores quedan almacenados en:

r _____

s _____

t _____

u _____

6. Suponga que a = 15; b = 18; c = 20.

```
public void probar(int a, int b, int c) {
        boolean r=false;
        r = ((a > b)||(a < c)) && ((a == c) || (a >= b));
        System.out.println(r);
}
```

¿Cuál será el resultado de r? _____

7. Suponga que: a = 15; b = 18; c = 20; d = 7.

```
public void probar(int a, int b, int c, int d) {
        boolean r=false;
        r = ((a >= b) || (a < d)) && (( a >= d) && (c > d));
        System.out.println(r);
}
```

¿Cuál será el resultado de r? _____

8. Suponga que: a = 15; b = 18; c = 20.

```
public void probar(int a, int b, int c) {
        boolean r=false;
        r = !(a == c) && (c > b);
        System.out.println(r);
}
```

¿Cuál será el resultado de r que se imprime? _____

9. Suponga que: a = 4; x = 9-

```
public void probar(int a, int x) {
        boolean b;
        b = (++a > ((x - a) -1)) || (++a != 4);
        System.out.println(b);
}
```

¿Cuál será el resultado de b? _____

¿Cuál será el resultado de a? _____

¿Cuál será el resultado de x? _____

10. Para comparación de cadenas de la siguiente información:

String moneda = "dollar";

String monedaInternacional = " Dollar"

Cuál será el resultado de la comparación:

moneda.equals(monedaInterancional) _____

moneda.equalsIgnoreCase(monedaInterancional) _____

moneda.compareTo(monedaInterancional) _____

moneda.equals("dólar") _____

"dollar".equalsIgnoreCase("Dollar") _____

moneda.compareTo("Dollar") _____

11. Implementar un método que permita determinar si un número es múltiplo de 3.
12. Diseñar el método que permita determinar si dos números son iguales o diferentes.
13. Elaborar una clase que implemente los métodos para las operaciones básicas: suma, resta, producto y división.
14. Diseñar una clase que provea los métodos para determinar el mayor de dos números enteros diferentes y el número menor.
15. Implementar un método para obtener el menor de tres números.
16. Construir una aplicación en Java que, a partir de la obtención de una fecha cualquiera (dd/mm/aaaa), imprima la fecha del día siguiente.

5.12 Bibliografía

Aguilar, L. J. (2004). *Fundamentos de programación algoritmos y estructura de datos*. Tercera edición. Ciudad: México. McGraw-Hill.

Booch, G. (1996). *Análisis y diseño orientado a objetos con aplicaciones*. Ciudad: México. Addison Wesley.

Deitel y Deitel (2012). *Cómo programar en Java*. Ciudad: México. Editorial Pearson (Prentice Hall).

Fowler, M. y Scott, K. (1999). *UML gota a gota*. Ciudad: México. Editorial Pearson.

Villalobos, J. y Casallas, R. (2006). *Fundamentos de programación, aprendizaje activo basado en casos*. Ciudad: México. Editorial Pearson.

Capítulo 6
Ciclos

6.1 Temática a desarrollar

- Estructuras de control repetitivas
- while
- do while
- for

6.2 Introducción

Los ciclos se encuentran entre las estructuras más comunes y utilizadas en programación, permitiendo que una o más instrucciones se repitan sin tener que volver a digitar el mismo código. Los ciclos o iteraciones en Java son:

- Ciclo while: se utiliza para ejecutar una serie de instrucciones (o una sola) mientras se cumpla una condición; cuando esta se deje de cumplir, las instrucciones que están dentro del ciclo dejarán de ejecutarse.
- Ciclo do-while: es bastante similar al ciclo while, con la única diferencia de que en el ciclo while, si no se cumple la condición desde la entrada, no se realizarán las instrucciones, pero en el caso del do-while, siempre se ejecutan las instrucciones de su bloque como mínimo una vez.
- Ciclo for: permite ejecutar un conjunto de sentencias un número determinado de veces fijado al principio del ciclo.
- Ciclo for mejorado (for-each): es la estructura iterativa ideal para recorrer colecciones de objetos o de cualquier otro tipo de valor desde el inicio hasta el final.

El uso de una u otra de estas estructuras de control depende de las preferencias del programador y, en muy pocos casos, del enunciado del problema, teniendo presente que unas pueden tener mejor ventaja que otra.

La figura 21 del mapa mental de ciclos complementa el concepto de las iteraciones en Java.

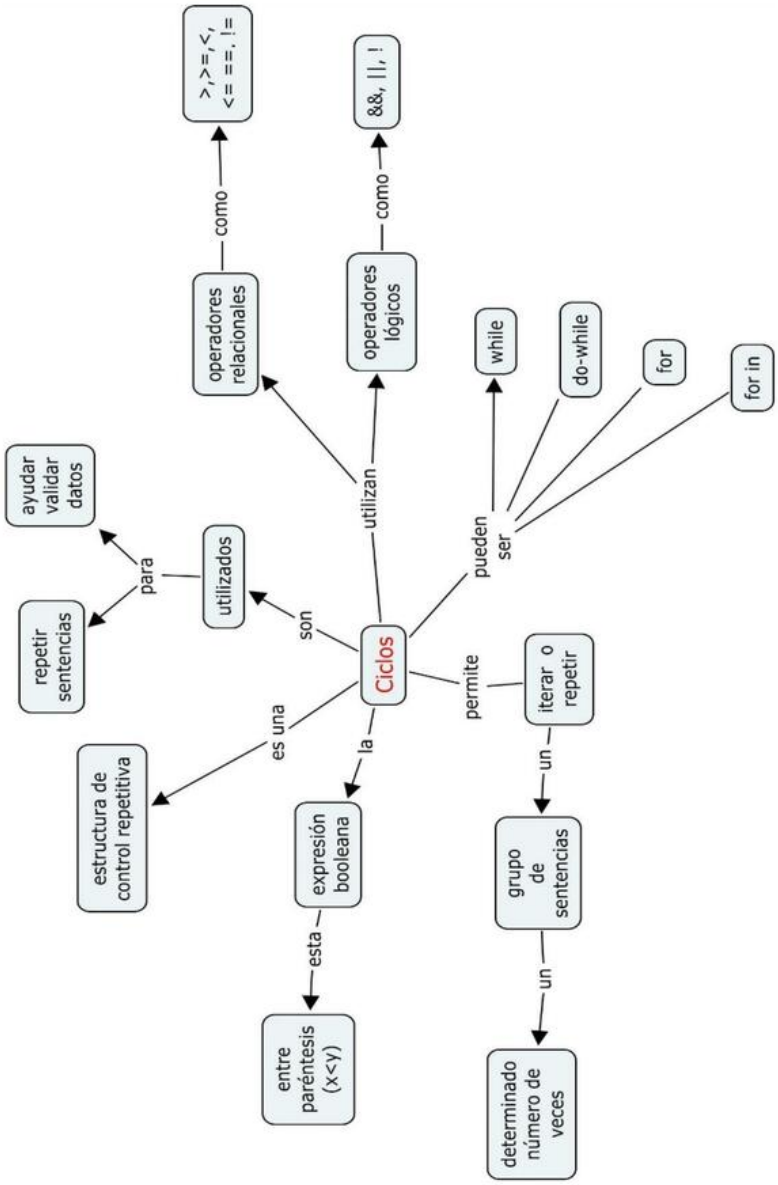


Figura 21.
Mapa mental de ciclos

6.3 Estructuras de control repetitivas o ciclos

Son estructuras iterativas que permiten repetir una o varias sentencias un número determinado de veces, utilizadas cuando se hace necesario iterar el mismo conjunto de instrucciones una cantidad específica de veces, determinada por una expresión booleana en programa o método.

La estructura de un ciclo comprende:

- Una variable o más variables de control o variables iniciales.
- Una condición booleana de parada o fin del ciclo.
- Un incremento o decremento de la variable de control.

Los tipos de ciclos que se pueden utilizar en un programa Java son:

- While
- do while
- for
- for-each o for mejorado

6.3.1 Estructura de repetición while (mientras)

Estructura repetitiva utilizada en programación que permite especificar que un programa o método debe repetir un bloque de código mientras se cumpla la condición; cuando deje de cumplir la condición, las iteraciones terminan y se ejecutará la primera sentencia que está inmediatamente después de la estructura repetitiva while.

La sintaxis general de esta instrucción while es:

```
while (expresiónBooleana) {
        Instrucción1;
}
```

Se recomienda utilizar las llaves { } cuando es una sola sentencia; para el caso en que se tenga más de una instrucción, es obligatorio.

```
while (condición) {
        Instrucción1;
        Instrucción2;
    ...
        InstrucciónN
}
```

```
while (condición) {
        Instrucción1;
}
```

La estructura repetitiva while evalúa la condición o expresión booleana para determinar si itera el conjunto de instrucciones que están entre las llaves; mientras la condición sea verdadera, se repite el bloque de instrucciones que se encuentre entre los paréntesis cursivos.

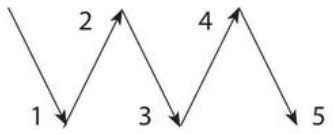
Para el funcionamiento del ciclo while, se plantea la siguiente situación: se requiere generar la impresión de los cinco primeros números naturales.

Acción	Proceso	Impresión del número
Inicialización	numero = 1;	
Impresión	System.out.println(n);	1
Incremento	numero ++;	
Impresión	System.out.println(n);	2
Incremento	numero ++;	
Impresión	System.out.println(n);	3
Incremento	numero ++;	
Impresión	System.out.println(n);	4
Incremento	numero ++;	
Impresión	System.out.println(n);	5
Incremento	numero ++;	

Como se puede observar, las instrucciones que se repiten son la impresión y el incremento de la variable numero; para este caso, estas serían las instrucciones que se incluirían dentro del ciclo while.

inicialización;
 	
while(condicion) {
instrucciones a repetir;
incremento o
decremento
}
	Inicialización int numero = 1;
	Condición o expresión booleana, numero <= 5
	expresiones a repetir; System.out. println(n);
	incremento numero++;

while (condición o expresión boolena) {



Número de iteraciones

}

Lo ideal es utilizar el ciclo while cuando se conoce previamente el número de iteraciones.

```
//inicialiación
numero = 1;
while (numero <= 5) {//condición numero <= 5
  //intrucciones que se repiten
        System.out.println(""+numero); //impresión
  numero++; //incremento
}
```

El comportamiento de la anterior rutina es:

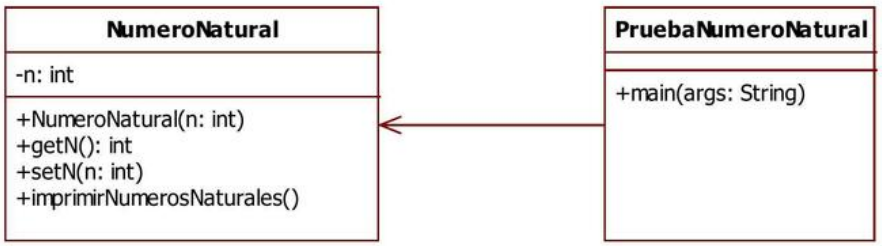
- Mientras la condición (numero <= 5) se cumpla, se realizarán las sentencias definidas entre el bloque de las llaves { ... }: System.*out*.println(""+numero); y numero++;.
- Cada vez que se termina de ejecutar el bloque de instrucciones, se vuelve a evaluar la condición; si es verdadera, nuevamente se vuelve a realizar el bloque de instrucciones que están entre las llaves.

- Se termina la ejecución cuando la condición pasa a ser falsa, que para el caso es cuando la variable numero es mayor que cinco.

Ejemplos:

1. Programa que imprime los diez primeros números naturales.

Diagrama de clases:



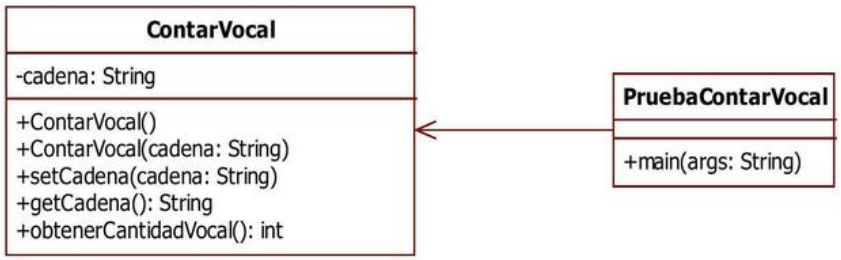
Codificación en Java del modelado del problema a partir del diagrama de clases:

```
public class NumeroNatural {
        private int n;

        public NumeroNatural(int n) {
                this.n = n;
        }
        public int getN() {
                return n;
        }

        public void setN(int n) {
                this.n = n;
        }

        public void imprimirNumerosNaturales() {
                int x = 1;
                 /**
                  * En el ciclo while se evalúa x mientras que x <= n
                  * se imprime el valor de x e incrementa en uno
```

```
                */

                while (x <= n)  {
                        System.out.println(" "+ x);
                        x ++;
                }
        }
}

import java.util.Scanner;
public class PruebaNumeroNatural {
        public static void main(String[] args) {
                Scanner lea = new Scanner(System.in);
                int numero = 0;
System.out.print("Digite la cantidad de números
naturales a imprimir: ");
                numero = lea.nextInt();
                NumeroNatural unNumero = new NumeroNatural(numero);
                unNumero.imprimirNumerosNaturales();
        }
}
```

2. Programa que cuenta el número de veces que aparece la vocal "e" en una cadena determinada.

Diagrama de clases:



Codificación en Java del modelado del enunciado a partir del diagrama de clases:

```
public class ContarVocal {
        /**
```

```
        * atributo de la clase
        */
        private String cadena;

        /**
        * constructor que inicializa la cadena con un espacio
*en blanco
        */
        public ContarVocal( ) {
                this.cadena = "";
        }

        /**
        * Método que actualiza el contenido de la cadena
        * @param cadena
        */
        public void setCadena(String cadena) {
                this.cadena = cadena;
        }

/**
        * Método que retorna el valor de la cadena
        * @return cadena
        */
        public String getCadena( ) {
                return cadena;
        }

        /**
        * Método que cuenta el número de apariciones de la vocal e
        */
        public int obtenerCantidadVocal( ) {
                int i = 0;
                int cantidad = 0;
                while (i < cadena.length()) {
                        if(cadena.charAt(i)=='e') {
                                cantidad ++;
                        }
                        i++;
                }
                return cantidad;
```

```
        }
}

import java.util.Scanner;

public class PruebaContarVocal {
        public static void main(String[] args) {
                ContarVocal cuentaVocal = new ContarVocal();
                String cadena="";
                int cantidad = 0;
                Scanner lea = new Scanner(System.in);
                System.out.print ("Digite Cadena: ");
                cadena = lea.nextLine();
                cuentaVocal.setCadena(cadena);
                cantidad = cuentaVocal.obtenerCantidadVocal();
                System.out.println("La vocal e esta presente "+ cantidad
      + " veces"+ " en la cadena: " +cuentaVocal.getCadena());

        }
}
```

Sentencias break y continue

Estas sentencias cambian el flujo de control especialmente en los ciclos o estructuras repetitivas.

La sentencia break permite que un programa se salga de los ciclos while, for, do-while y de la sentencia de selección switch en el cual se encuentra.

La sentencia break permite salir de las estructuras de control anidadas, pasando el control del programa a la instrucción que se encuentra luego de la llave de cierre del bloque.

```
while ( expresiónBoolean) {
        while ( expresiónBoolean) {
                if( condicion ) {
                        break;
```

```
                }
        }
}
```

La sentencia continue procede con la siguiente iteración (repetición) del ciclo while, for o do-while, saltándose las sentencias que se encuentran luego de la sentencia continue.

```
while ( expressionBoolean) {
        if(expresiónBoolean ) {
                continue;
        }
}
```

6.3.2 Estructura de repetición do-while

El ciclo do-while ejecuta una instrucción o un bloque de instrucciones que están entre las llaves { } mientras que la expresión booleana especificada en el while se evalúe como verdadera. La sintaxis general es:

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

```
do {
        instrucciones a repetir;
        incremento o decremento
} while (condicionBooleana);
```

La diferencia del significado de do-while respecto a while es que, en el do-while, primero ejecuta las instrucciones y luego evalúa la condición o expresión booleana y de esta forma podrá seguir iterando las instrucciones que están dentro del ciclo.

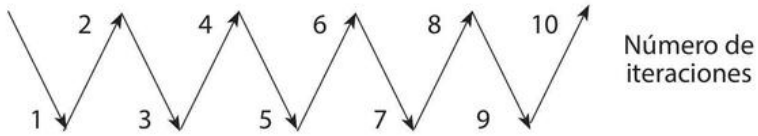
```
int numero = 1;
do{
 System.out.println(""+numero);
      numero++;
} while (numero < 11);
```

Inicialización variable de control

Incremento variable de control

Condición

```
  int numero = 1;
 do{
```




```
 } while (numero < 11);
```

Ejemplos:

1. Aplicación que genera la impresión de los n primeros números naturales, utilizando el ciclo do-while.

```
public class NumeroNatural {
        private int n;

        public NumeroNatural(int n) {
                this.n= n;
        }

        public void setN(int n) {
                this.n = n;
        }

        public int getN() {
```

```
            return n;
        }

        public void imprimirNumeros() {
            int x = 1;
            do {
                System.out.print(" "+x);
                x++;
            } while (x <= n);
        }
}
```

```
import java.util.Scanner;
public class PruebaNumeroNatural {
        public static void main(String[ ] args) {
          int cantidad = 0;
          Scanner lea = new Scanner(System.in);
          System.out.print("Cantidad de números naturales a imprimir: ");
          cantidad = lea.nextInt();
          NumeroNatural numero = new NumeroNatural(cantidad);
          numero.imprimirNumeros();
        }
}
```

2. Programa que cuenta el número de veces que aparece la vocal "e" en una cadena determinada, haciendo uso del ciclo do-while.

```
public class ContarVocal {

        /**
         * atributo de la clases
         */
        private String cadena;

        /**
         * constructor que inicializa la cadena con un espacio en blanco
         */
        public ContarVocal( ) {
            this.cadena = "";
```

```
        }

        /**
         * Método que actualiza el contenido de la cadena
         * @param cadena
         */
        public void setCadena(String cadena) {
                this.cadena = cadena;
        }



        /**
         * Método que retorna el valor de la cadena
         * @return cadena
         */
        public String getCadena( ) {
                return cadena;
        }

        /**
         * Método que cuenta el número de apariciones de la vocal e
         */

        public int obtenerCantidadVocal( ) {
                int i = 0;
                int cantidad = 0;
                do {
                        if(cadena.charAt(i)=='e') {
                                cantidad ++;
                        }
                        i++;
                } while (i < cadena.length());
                return cantidad;
        }
}
```

La clase PruebaContarVocal es similar al ejemplo anterior; nótese que en la clase ContarVocal los métodos constructores set y get son idénticos.

6.3.3 Estructura de repetición for

Utilizada para repetir una instrucción o un conjunto de instrucciones hasta que una condición se cumpla. La sintaxis de la sentencia for es:

```
for (instrucción; condición; instrucción)  {
        instrucciones
}

for (inicialización; condición; incremento)  {
        instrucciones
}

for (inicialización variable ; expresiónBooleana; incremento o decremento) {
  Instrucción (es) o acciones;
}
```

Inicialización: es una instrucción que permite dar valor inicial a la variable de control del ciclo, por ejemplo, int n = 1.

Expresión booleana: es una condición que se evalúa antes de realizar el bloque de instrucciones en el ciclo. Esta condición determina cuándo se termina la ejecución del ciclo, por ejemplo, n < = 10.

Incremento: es una instrucción usada para incrementar o decrementar el valor de las variables de control, por ejemplo, n ++; n - -.

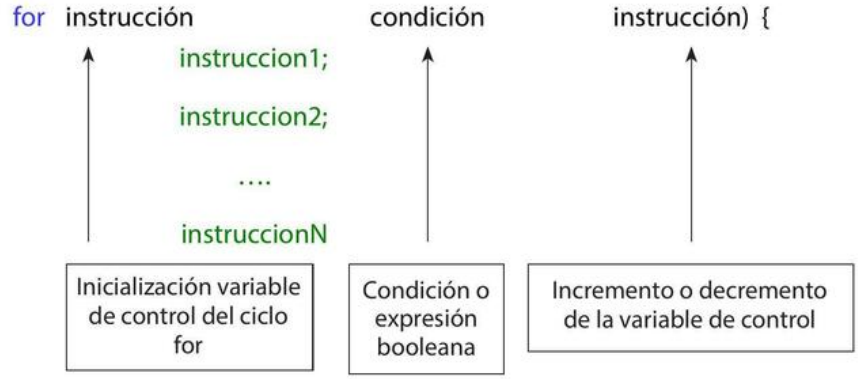
```
for (int n = 1; n < = 10; n ++)
for (int x = 10; n >= 1; n - -)
```

Funcionamiento del ciclo for

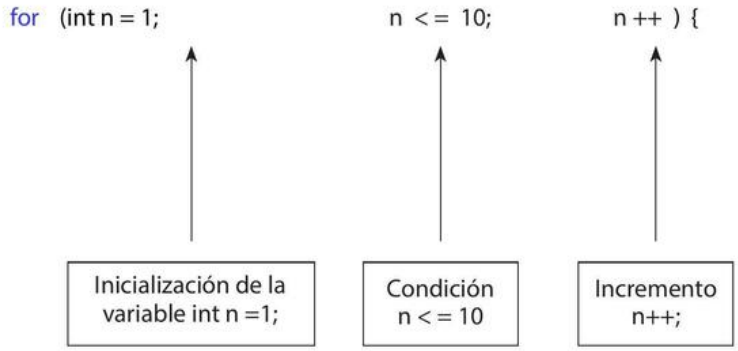
La estructura repetitiva del ciclo for está conformada por tres partes y un grupo de instrucciones que están entre las llaves { }.

```
for ( instrucción ; condición ; instrucción) {
instruccion1;
```

```
instruccion2;
….
instruccionn;
}
```

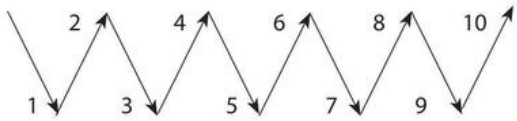


Impresión de los 10 primeros números naturales.




```
//instrucción

System.out.println(""+n);

}
```

```
for (int n = 1; n <= 10; n ++) {
```



Ejecución de las instrucciones o bloque de instrucciones que están dentro de las llaves.

```
}
```

- La inicialización (int n = 1;) se realiza una única vez, antes de la primera iteración.
- La condición (n < = 10;) se revisa antes de comenzar una iteración.
- El incremento o decremento de la variable de control (n ++) se realiza cada vez que termina una iteración.

Ejemplos:

1. Programa que imprime los diez primeros números naturales utilizando el ciclo for.

```
/**
* Clase que implementa la impresión de los n primeros números naturales
* utilizando el ciclo for
*/

public class NumeroNatural {

        private int n;

        public NumeroNatural(int n) {
                this.n = n;
        }
        public int getN() {
                return n;
        }

        public void setN(int n) {
                this.n = n;
        }
```

```
        public void imprimirNumerosNaturales()
        {
                /**
                 * En el ciclo for se evalúa x mientras que x <= n
                 * se imprime el valor de x e incrementa en uno
                 */
                for (int x = 1; x <= n; x++) {
                        System.out.println(" "+ x);

                }

        }
}
```

La clase PruebaNumeroNatural es idéntica al ejemplo anterior; nótese que en la clase NumeroNatural los métodos constructor set y get son idénticos.

2. Programa que cuenta el número de veces que aparece la vocal "e" en una cadena determinada haciendo uso del ciclo for.

```
public class ContarVocal {
        /** atributo de la clases */
        private String cadena;
        /**constructor que inicializa la cadena con un espacio en blanco */
        public ContarVocal( ) {
                this.cadena = "";
        }
        /**
         * Método que actualiza el contenido de la cadena */
        public void setCadena(String cadena) {
                this.cadena = cadena;
        }


        /**
         * Método que retorna el valor de la cadena */
        public String getCadena( ) {
                return cadena;
        }
```

```
/** Método que cuenta el número de apariciones de la vocal */
        public int obtenerCantidadVocal( ) {
                int cantidad = 0;
                for (int i = 0; i < cadena.length(); i++) {
                        if(cadena.charAt(i)=='e') {
                                cantidad ++;
                        }
                }
                return cantidad;
        }

}
```

La clase PruebaContarVocal es idéntica al ejemplo anterior; nótese que en la clase ContarVocal los métodos constructor set y get son iguales a los del ejemplo del ciclo while.

6.4 Comparación de los ciclos while, do-while, for

Los tres ciclos son estructuras repetitivas, como se puede observar en el ejemplo del programa que imprime los diez primeros números naturales.

```
    Ciclo while
    public class NumeroNaturalWhile{
Ciclo for
public class NumeroNaturalFor {
   public void imprimirNumero ()    {
   //inicialiación;   condición;
incremento
 for (int numero = 1;  numero <= 10;
numero++)  {
                //intrucción que se repite
        System.out.println(""+numero);
//impresion
       }
   }
}
```

Diagrama de clases

Se puede utilizar el mismo nombre de la clase, ya que los nombres del método son diferentes.

Los métodos imprimen los diez primeros números naturales 1, 2, 3, …, 10.

Todos tienen:

- Inicialización (numero = 1)
- Condición (numero < =10)
- Incremento (numero ++)

```
        do{
  System.out.println(""+numero);
            numero++;
    } while (numero <= 10);
  }
}
```

En el ciclo while, se evalúa la condición y después se ejecutan las instrucciones de impresión e incremento.

En el ciclo do-while, se ejecutan las instrucciones de impresión e incremento y después se evalúa la condición.

En el ciclo for, inicialización del número en 1; se evalúa la condición; se ejecutan las instrucciones de impresión y después se realiza el incremento.

6.5 Ciclo for-each (for mejorado)

Es un ciclo repetitivo que no utiliza un contador. Su uso es indicado para colecciones de objetos, como, por ejemplo: arreglos, matrices, listas, entre otras estructuras que no se incluyen en este texto.

Ejemplo:

Programa que imprime los días de la semana

```
public class DiaSemana {
        public static void imprimirDiasSemana( ) {
                String semana [ ] = {"Lunes", "Martes", "Miércoles",
"Jueves", "Viernes", "Sábado", "Domingo"};
                for (String dia: semana){
                 System.out.print(" "+dia);
                }
        }

        public static void main(String[] args) {

                imprimirDiasSemana();
        }
}
```

6.6 Uso de los ciclos repetitivos

Determinar cuándo utilizar uno u otro ciclo repetitivo es un proceso complejo porque existen equivalencias entre los ciclos; lo que hay son recomendaciones y nociones generales de cuando usar alguno de ellos:

- Un ciclo for se puede utilizar cuando se conoce cuántas iteraciones son necesarias realizar y, sobre todo, cuando la variable que hace las veces de contador no debe ser modificada.

- El ciclo while, cuando la variable de control de ciclo debe incrementarse/decrementarse según una o varias condiciones.

- Un ciclo do..-while, cuando se necesita ejecutar las sentencias del ciclo repetitivo por lo menos una vez.

- El for-each, para realizar tareas de consulta de datos y no para asignación de valores.

6.7 Lecturas recomendadas

- Estructuras repetitivas o ciclos.
- Establecer la principales diferencias entre las siguientes versiones del ciclo for:

```
inicializadores;
for( ; condiciones; incrementos) {
instrucciones;
}
```

```
inicializadores;
for( ; condiciones; ) {
instrucciones;
incrementos;
}
```

```
inicializadores;
for( ; ; ) {
// debe tener un break para romper el ciclo
```

```
instrucciones;
incrementos;
}
```

6.8 Preguntas y ejercicios de revisión de conceptos

1. Importancia de los ciclos en Java.
2. Realizar cuadro comparativo con las semejanzas y diferencias de los ciclos.
3. Cómo identificar que se debe aplicar un ciclo para resolver un problema.

6.9 Ejercicios

1. Establezca la diferencia entre contador y acumulador.
2. Aplicación que genera la impresión de los 10 primeros números pares.
3. Implementar un método que imprima los n primeros múltiplos del 7.
4. Diseñar el método que imprima la sumatoria de los n primeros números impares.
5. Generar la impresión de cualquier tabla de multiplicar, donde el multiplicador vaya hasta 12.
6. Implementar una aplicación que permita contar el número de vocales de una cadena.
7. Las claves de acceso son la barrera más común para evitar el acceso no autorizado a un sistema, son un recurso informático de gran importancia para el acceso a una aplicación para realizar transacciones en un cajero electrónico. Se requiere de una aplicación que implemente una clave de acceso para identificar si la clave digitada corresponde con la clave que se tiene con antelación almacenada en un atributo.
8. Construir una aplicación que permita determinar el número de apariciones de las vocales, incluyendo mayúsculas y minúsculas, que existen en una cadena de caracteres, haciendo uso del método constructor con parámetros.

6.10 Bibliografía

Aguilar, L. J. (2004). *Fundamentos de programación algoritmos y estructura de datos*. Tercera edición. Ciudad: México. McGraw-Hill.

Barnes, D. y Kölling, M. (2007). *Programación orientada a objetos con Java*. Ciudad: Madrid. Prentice Hall.

Bertrand, M. (1999). *Construcción de software orientado a objetos*. Segunda edición. Ciudad: Madrid. Prentice Hall.

Booch, G. (1996). *Análisis y diseño orientado a objetos con aplicaciones*. Ciudad: México. Addison Wesley.

Deitel y Deitel. (2012). *Cómo programar en Java*. Ciudad: México. Editorial Pearson (Prentice Hall).

Fowler, M. y Scott, K. (1999). *UML gota a gota*. Ciudad: México. Editorial Pearson.

Graig, L. (2003). *UML y patrones: una introducción al análisis y diseño orientado a objetos y al proceso unificado*. Segunda edición. Ciudad: México. Prentice Hall.

Rumbaugh, J., Jacobson, I. y Booch, G. (2005). *The Unified Modeling Language User Guide*. Ciudad: San Francisco. Addison-Wesley, Reading, Mass. ISBN-13: 078-5342267976, ISBN-10: 0321267974.

Villalobos, J. y Casallas, R. (2006). *Fundamentos de programación, aprendizaje activo basado en casos*. Ciudad: México. Editorial Pearson.

Capítulo 7

Cadenas

7.1 Temática a desarrollar

- String
- Métodos asociados a la clase String

7.2 Introducción

Un String es una secuencia finita de caracteres, que pueden ser letras, dígitos, símbolos especiales; en Java existen varias clases para manipular cadenas de caracteres, una de estas es la clase String definida en la API de Java. Esta clase proporciona algunos métodos dentro de los que se permite convertir el objeto cadena en caracteres, convertir números en cadenas, buscar palabras en una cadena, crear substrings, cambiar la cadena de caracteres mayúsculas a minúsculas o viceversa, obtener la longitud de la cadena, comparar cadenas, entre otros. La figura 20 detalla el concepto de la clase String.

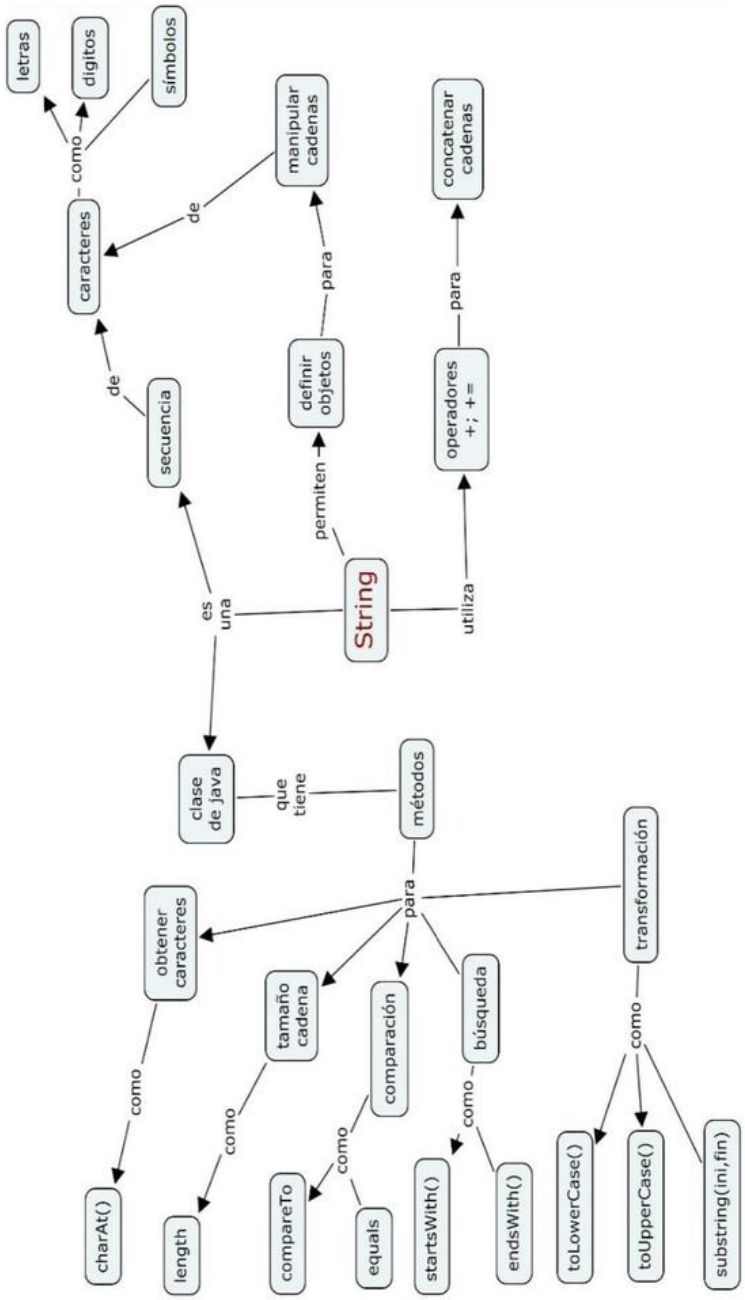


Figura 20.
Mapa mental de la clase String

7.3 Clase String

Un String en Java representa una cadena de caracteres no modificable. Así, por ejemplo, la instrucción String str ="Universidad"; define el String str y cada una de las letras que lo integran (U, n, i, v, e, r, s, i, d, a, d) y esta palabra no se puede modificar (inmutable).

El API de Java proporciona, para la clase String, los constructores y una amplia gama de métodos para el manejo de cadenas; la cadena debe estar entre comillas dobles.

Tabla 12.
Ejemplo de cadenas

Sintaxis	Ejemplos
Etapa de declaración: String identificador;	String nombre; String dato;
Asignación: identificador = " mensaje ";	String nombre = "Carlos"; String dato = "123456";

Una cadena es una agrupación del dato primitivo carácter o *char*. En el lenguaje Java, una cadena es una instancia de la clase String y ofrece la siguiente funcionalidad:

- Inicialización
- Asignación
- Acceso
- Comparación
- Búsqueda
- Verificación
- Conversión a tipos

Algunos métodos básicos incluidos en la clase String se citan y se describe su funcionalidad en la tabla 13.

Tabla 13.
Métodos básicos

Método	Funcionalidad
String() String(String str)	Dos de los constructores incorporados en la clase String permiten crear una cadena vacía y una cadena a partir de otra existente.
char charAt(int index)	Este método devuelve el carácter presente en la cadena en la posición dada por el parámetro.
Int length()	Este método devuelve la longitud de la cadena.
boolean equals(String str)	Este método permite responder si la cadena enviada como parámetro es igual a la cadena actual.
String toLowerCase()	Este método retorna la cadena convertida en letras minúsculas.
String toUpperCase()	Este método retorna la cadena convertida a letras mayúsculas.

La concatenación es el proceso de unir dos o más cadenas. Por ejemplo, se definen dos objetos, str1 y str2, de la clase String y se utiliza el operador + para concatenar cadenas de caracteres o strings:

```
String str1 = "Hola"
String str2 = " Mundo"
String salida = str1+str2
```

El objeto salida almacena "Hola Mundo".

Otra forma de realizar la concatenación de cadenas es utilizando el operador += para unir el nombre con el apellido:

```
String nombre="Juan Carlos ";
nombre+="Bustamante";
System.out.println(nombre);
```

Algunos métodos asociados a la clase String:

int length( ): método que retorna la longitud de una cadena (String), incluyendo espacios en blanco. La longitud siempre es una unidad mayor que el índice asociado al último carácter de la String.

```
String nombre = "Ricardo Quintero";
int longitud = nombre.length();
```

El valor de la variable longitud es 16, correspondiente al número de caracteres de la cadena nombre.

boolean equals(Object obj): método que compara el String con el objeto especificado. El resultado es true si y solo si el argumento es no nulo y es un objeto String que contiene la misma secuencia de caracteres. Para comparar dos cadenas, se tiene:

```
String cadena1="Telecomunicaciones";
String cadena2=new String("Telecomunicaciones");
if(cadena1.equals(cadena1)){
        System.out.println("cadenas de igual contenido");
} else {
        System.out.println("Cadenas Diferentes");
}
```

Rutinas que realizan la comparación de una cadena que conforma una clave de acceso, utilizando el método equals( ):

```
String password = "ABC123";
if ( password.equals("abc123") {
        System.out.println("Clave correcta");
} else {
        System.out.println("Clave errada");
}
```

Se imprime clave errada, ya que hay diferencias entre las minúsculas y mayúsculas.

boolean equalsIgnoreCase(String str): método que evalúa si dos String tienen los mismos caracteres y en el mismo orden sin tener en cuenta las mayúsculas. Si es así, devuelve true y si no, false.

Para la comparación de una cadena que conforma una clave de acceso, utilizando el método equalsIgnoreCase( ):

```
String password = "abc123";
if password.equalsIgnoreCase("ABC123") {
        System.out.println("Clave correcta ");
} else {
        System.out.println("Clave Errada ");
}
```

Se imprime clave correcta, ya que el método equalsIgnoreCase( ) ignora la diferencias entre las minúsculas y mayúsculas.

boolean startsWith(String str): método que retorna true si el String sobre el que se aplica inicia con el argumento; false si esto no ocurre.

El siguiente fragmento de código evalúa si la cadena inicia con los caracteres ip:

```
String parametro = "ip = 192.168.100.2";
if (parametro.startsWith("ip"))  {
        System.out.println("La direccion "+parametro);
} else  {
        System.out.println("El parámetro no es una ip");
}
```

boolean endsWith(String str): método que retorna true si el String termina en los caracteres indicados; false si esto no ocurre.

Líneas de código que evalúa si la cadena termina con .co:

```
String url = "http://www.ulibertadores.edu.co";
if (url.endsWith(".co"))  {
        System.out.println("Pagina colombiana");
} else {
        System.out.println("Pagina extranjera");
}
```

char charAt(int indice): método que retorna el carácter en la posición indicada por índice. El rango de índice va de 0 a length() - 1. cadena.charAt(i).

Ejemplos:

1. Ubicar un carácter mediante un índice.

```
String nombre = "Miguel Hernández";
char c = nombre.charAt(0);
```

2. Se imprime la M, correspondiente al primer carácter.

Impresión de todos los caracteres de una cadena.

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

String toLowerCase(): método que convierte un cadena a letras minúsculas.

Ejemplos:

1. Se almacena una cadena de caracteres en el objeto cadena1 y se imprime en letras mayúsculas:

```
String cadena1="Telecomunicaciones";
System.out.println(cadena.toLowerCase());
```

La impresión es TELECOMUNICACIONES.

2. Se almacena en el objeto titulo una cadena de caracteres y se realiza la conversión a mayúsculas, asignando el resultado a la cadena mayúscula.

```
String titulo = "Fundación Universitaria";
String mayuscula = titulo.toUpperCase();
```

El resultado es:
mayuscula = "FUNDACIÓN UNIVERSITARIA";

String toUpperCase(): método que convierte una cadena a letras mayúsculas.

1. Se almacena una cadena de caracteres en el objeto cadena1 y se imprime en letras minúsculas:

   ```
   String cadena1="Telecomunicaciones";
   System.out.println(cadena1.toUpperCase());
   ```

   La impresión es TELECOMUNICACIONES.

2. Se almacena en el objeto titulo una cadena de caracteres y se realiza la conversión a mayúsculas, asignando el resultado a la cadena minúscula:

   ```
   String titulo = "Fundación Universitaria";
   String minuscula = titulo.toLowerCase();
   minuscula = "Fundación Universitaria";
   ```

   El resultado es:
   minuscula = "Fundación Universitaria";

String substring(int inicial, int fin): método que retorna una parte o subcadena (substring) de un String dado.

1. Se extrae una subcadena a partir de la posición inicial y una final:

   ```
   String nombre = "Oscar Alejandro";
   String subCadena = nombre.substring(6,14);
   ```

   La variable subCadena almacena "Alejandro", ya que corresponde al sexto carácter y va hasta el 14.

2. Se extrae de un substring especificando la posición de comienzo y la del final.

   ```
   String str = "El lenguaje Java";
   ```

```
String subStr=str.substring(3, 11);
```

La variable subStr almacena la subcadena "lenguaje".

String trim( ): método que retorna la cadena (String) que se le pasa al argumento, pero sin espacios en blanco al principio ni al final. No elimina los espacios en blanco situados entre las palabras.

```
String autores = " Baquero & Hernández ";
String resaltar = "*"+autores.trim()+"*";
```

Baquero & Hernández elimina el espacio al inicio y le agrega asterisco al inicio y final de la cadena.

String replace (char anteriorChar, char nuevoChar): método que permite cambiar el carácter asociado al primer argumento por el segundo argumento, de la cadena (String) del método, generando una nueva cadena. La cadena (String) sobre la que se aplica el método replace no cambia, simplemente se crea otra cadena en base a la String sobre la que se aplica el método, tal como se puede observar en las líneas de código presentadas a continuación.

```
// Cadena sobre la que se realiza la sustitución
String cadena1 = "Institucion Universitaria Los Libertadores";

// Cadena en la que se almacena el resultado
String cadena2 = null;

cadena2 = cadena1.replace("Institución","Fundación");

System.out.println(cadena2);
// Fundación universitaria los Libertadores

System.out.println(cadena1);
/* Al ejecutar el método
replace no afecta a la cadena original */
```

7.4 Obtener cadenas desde las primitivas

La clase **String** tiene métodos implementados para transformar valores de otros tipos de datos a su representación como cadena. Todas estas funciones tienen el nombre de *valueOf*, estando el método sobrecargado para todos los tipos de datos básicos.

Rutina que convierte datos numéricos a cadenas (String):

```
String tres = String.valueOf(3);
String dosPuntoUno = String.valueOf(2.1);
final double PI = 3.141592;
String piString = String.valueOf(PI);
```

7.5 Obtener primitivas desde las cadenas

Para esto se utilizan los métodos de las clases Integer y Float.

```
String dato = "2011";
int alfaInteger = Integer.parseInt(dato);
String beta = "20.11";
float betaFloat = Float.parseFloat(beta);
```

compareTo(String str): método que realiza una comparación entre un String (en este caso) y otro objeto de tipo String, retornando un número entero menor que 0 si la cadena argumento es lexicográficamente mayor que la cadena a comparar, 0 si son iguales y será mayor que 0 si es menor.

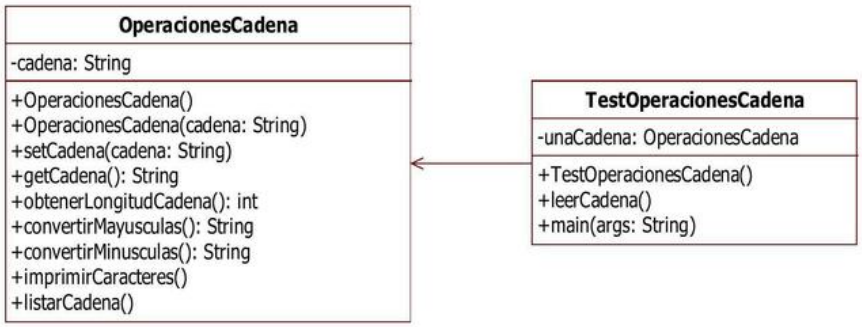
```
String str1 = "a";
String str2 = "b";

System.out.println(str1.compareTo(str2)); // -1
System.out.println(str2.compareTo(str1)); // 1  str12 > str1
```

```
str2 = "aa";

System.out.println(str1.compareTo(str2)); // -1
System.out.println(str2.compareTo(str1)); // 1

str2 = "c";

System.out.println(str1.compareTo(str2)); // -2
System.out.println(str2.compareTo(str1)); // 2

str1 = "c";

System.out.println(str1.compareTo(str2)); // 0
System.out.println(str2.compareTo(str1)); // 0  cadena iguales
```

Programas de ejemplo:

1. Aplicación que, a partir de una cadena ingresada por teclado, imprime: los caracteres que la conforman, el tamaño de la cadena y la conversión a mayúsculas y minúsculas.

El siguiente diagrama de clases define unas operaciones básicas de Strings.




```
/**
 * clase que modela las operaciones de una cadena como:
 *  tamaño de la cadena, convertir a mayúsculas y minúsculas.
 * @author  Ing. Miguel  Hernández Bejarano
 * @version 1.0
 */
```

```
public class OperacionesCadena {
        /**
         * atributo de la clases
         */
        private String cadena;

        /**
          * constructor que inicializa la cadena con un espacio
  * en blanco
         */
        public OperacionesCadena( ) {
                this.cadena = "";
        }


        /**
         * Constructor que permite crear cadenas
         * @param cadena
         */
        public OperacionesCadena(String cadena) {
                this.cadena = cadena;
        }


  /**
         * Método que actualiza el contenido de la cadena
         * @param cadena
         */
        public void setCadena(String cadena) {
                this.cadena = cadena;
        }

        /**
         * Método que retorna el valor de la cadena
         * @return cadena
         */
        public String getCadena( ) {
                return cadena;
        }

        /**
```

```
        * Método que retorna el tamaño o longitud de la cadena
        * @return tamaño cadena
        */
        public int obtenerLongitudCadena( ){
                return cadena.length();
        }
        /**
        * Método que convierte y retorna la cadena en mayúsculas
        * @return cadena en mayúsculas
        */
        public String convertirMayusculas( ) {
                return cadena.toUpperCase();
        }

    /**
        * Método que convierte y retorna la cadena en minúsculas
        * @return cadena en minúsculas
        */

        public String convertirMinusculas( ) {
                return cadena.toLowerCase();
        }
        /**
        * Método que imprime los caracteres que conforman la cadena
        */
        public void imprimirCaracteresCadena( ){
                char c='';
                System.out.println("\nImpresión de la Cadena por
caracteres");
                for(int i=0;i<cadena.length();i++){
                        c=cadena.charAt(i);
                        System.out.println(i+" "+c);
                }
                System.out.println(" ");
        }

        /**
        * Método que imprime la cadena
        */
        public void imprimirCadena( ){
                System.out.println(" "+cadena);
```

```
        }
}



import javax.util.Scanner;
/**
 * clase que modela la lectura de la cadena e
 * impresión de los resultados
 * @version 1.0
 */
public class Cadena {
        /**
         * atributos se define objeto de la clase OperacionesCadena
         */
        private OperacionesCadena operacion;

        /**
         * constructor de la clase cadena
         */
        public Cadena(){
                this.cadena="";
                this.operacion= new OperacionesCadena();
        }

        /**
         * método en el que se realiza la lectura de la cadena
         * e impresión de los resultados de operaciones con la cadena
         */
        public void leerCadena(){
                String cadena;
                String resultados = "";
                int tam = 0;
            Scanner in = new Scanner(System.in);
                //se lee la cadena por teclado
                cadena=in.nextLine();
System.out.println ("Digite Cadena ");

                System.out.println("Impresión de la cadena ");
                operacion.imprimirCaracteres(cadena);
                tam=operacion.obtenerlongitudCadena(cadena);
```

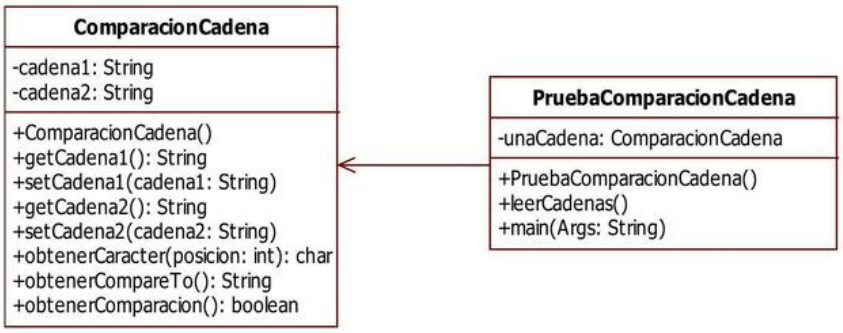
```
                System.out.println("\nLa Longitud de la cadena:" +
tam+" caracteres ");
                System.out.println("\nLa cadena texto en mayúsculas:"
+operacion.convertirMayusculas(cadena));
        System.out.println("La cadena texto en
minúsculas:" +operacion.convertirMinusculas(cadena));
        }


        /**
         * método principal, para iniciar la ejecución del proyecto
         * @param args
         */
        public static void main(String[] args) {
                //definción del objeto str de la clase Cadena
                Cadena str=new Cadena();
                str.leerCadena();
        }
}
```

- Crear un proyecto en Eclipse llamado cadena, copiar la clase OperacionesCadena y digitalizar la clase TestOperacionesCadena, compilar y correr la aplicación.

```
import java.util.Scanner;
/**
 * clase que modela las operaciones, la lectura de la cadena e
 *  impresión de los resultados de las operaciones de la cadena
 *  @ File Name : TestOperacionesCadena.java
 *  @author : Miguel Hernández
 *  @version 1.0
 */
public class TestOperacionesCadena {
        private OperacionesCadena unaCadena;
        public TestOperacionesCadena() {
                this.unaCadena= new OperacionesCadena();
        }
        public void leerCadena(String cadena){
          unaCadena.setCadena(cadena);
```

```
                System.out.println("\nLa Longitud de la cadena:" +
unaCadena.obtenerLongitudCadena()+" caracteres");
                System.out.println("\nLa cadena texto en mayúsculas:" +
unaCadena.convertirMayusculas());
                System.out.println("La cadena texto en minúsculas:" +
unaCadena.convertirMinusculas());
                unaCadena.imprimirCaracteresCadena();
                unaCadena.imprimirCadena();
        }

        public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
                TestOperacionesCadena prueba = new
    TestOperacionesCadena();
    String cadena="";
                //se lee la cadena por teclado
    System.out.println ("Digite Cadena");
                cadena=in.nextLine();

                prueba.leerCadena(cadena);
        }
}
```

2. Aplicación que, a partir de dos cadenas ingresadas por teclado, efectúa la comparación de las dos cadenas utilizando los métodos compareTo() y equals(), así como la impresión del segundo carácter de la cadena2.



```
public class ComparacionCadena {

        private String cadena1;
        private String cadena2;

        public ComparacionCadena( ) {
                this.cadena1 = "Hola";
                this.cadena2 = "ola";
        }


        public String getCadena1() {
                return cadena1;
        }

        public void setCadena1(String cadena1) {
                this.cadena1 = cadena1;
        }

        public String getCadena2() {
                return cadena2;
        }

        public void setCadena2(String cadena2) {
                this.cadena2 = cadena2;
        }

        /**
         * método que retorna el caracter de la segunda posición de
         * la cadena
         */
        public char obtenerCaracter(int posicion){
                char caracter=cadena2.charAt(posicion);
                return caracter;
        }

        /**
         * método que compara dos cadenas
         * @return retorna cero si son iguales
         */
```

```
	public String obtenerCompareTo(){
		int res=cadena1.compareTo(cadena2);
		String respuesta="";
		if (res==0)
			respuesta="Las cadenas son iguales";
		else if (res>0)
			respuesta="La cadena mayor es "+cadena1;
		else
			respuesta="La cadena mayor es "+ cadena2;
		return respuesta;
	}

	/**
	 * Método que compara dos cadenas
	 * @return true si las dos cadenas son iguales o false en caso
 * contrario
	 */
	public boolean obtenerComparacion(){
		if (cadena1.equals(cadena2)){
			return true;
		}
		else{
			return false;
		}
	}
}


import java.util.Scanner;

public class PruebaComparacionCadena {

	private ComparacionCadena unaCadena;

	/**
	 * contructor
	 */
	public PruebaComparacionCadena() {
		this.unaCadena = new ComparacionCadena();

	}
```

```
/**
 * método que realiza la lectura de las cadenas
 * e impresión de los resultados
 */

public void leerCadenas(){
        String cadena1;
        String cadena2;
        Scanner lea = new Scanner(System.in);
        char caracter = ' ';
        boolean cadenasIguales = false;
        String comparaCadenas = "";
        String cambioLetras = "";
        String segundaComparacion = "";


        //se lee la primera cadena
        System.out.println("Digite Primera Cadena ");
        cadena1=lea.nextLine();
        System.out.println("Digite Segunda Cadena ");
        //se lee la segunda cadena
        cadena2=lea.nextLine();

        this.unaCadena.setCadena1(cadena1);
        this.unaCadena.setCadena2(cadena2);


        //el objeto operación accede al método  CharAt()
        //que retorna el carácter de una determinada posición

        caracter=unaCadena.obtenerCaracter(1);

        comparaCadenas= unaCadena.obtenerCompareTo() ;

        cadenasIguales=unaCadena.obtenerComparacion();
        if(cadenasIguales==true){
                segundaComparacion="Cadenas Iguales ";
        }
        else{
                segundaComparacion="Cadenas diferentes ";
```

```
                }

                //impresión de resultados
                System.out.println("Primera Cadena:
"+unaCadena.getCadena1()+"\n"+
                "Segunda Cadena: "+unaCadena.getCadena2()+"\n"+
                "Caracter de la posición dos  de la Cadena1:"+caracter+"\n"+
                "Comparación de Cadenas método compareTo:
"+comparaCadenas+"\n"+
                "Segunda Comparación de Cadenas con el método equals():
"+segundaComparacion+"\n");
        }

        /**
         * método principal donde se ejecuta el proyecto
         * @param Args
         */
        public static void main(String Args []){
                PruebaComparacionCadena        prueba        =        new
PruebaComparacionCadena();
                prueba.leerCadenas();
        }

}
```

7.6 Lecturas recomendadas

- Clase StringBuffer.
- Clase StringReader.
- Clase Character.
- Clase StringTokenizer.

7.7 Preguntas de revisión de conceptos

- Establecer cuáles son las semejanzas y diferencias entre los métodos equals y compareTo.
- Describa las ventajas que se tienen sobre la utilización de cadenas.
- Determinar otro tipo de aplicación de las cadenas.

7.8 Ejercicios

Crear una clase que incluya los elementos incorporados a partir del siguiente String definido: String cadena = "Universidad".

U	n	i	v	e	r	s	i	d	a	d

Completar la sentencia de los métodos según el requerimiento planteado.

EjCadena
-cadena: String
+EjCadena( ) +imprimirPrimerElementoCadena( ) +imprimirTercerElementoCadena ( ) +imprimirUltimoElementoCadena( ) +imprimirPosicionesImpares ( ) +imprimirElementoMitad ( ) + imprimirUltimoElemento ( ) + imprimirCadena ( )

Imprimir el primer dato de la cadena
public void imprimirPrimerElemento-Cadena ( ) {

}

Imprimir el tercer elemento de la cadena
public void imprimirTercerElementoCadena ( ) {

}

ir al último dato de la cadena
public void imprimirUltimoElementoCadena ( ) {

}
Imprimir todos los elementos de las posiciones impares de la cadena (sugerencia: tenga en cuenta que este método debe incluir un ciclo).
public void imprimirPosicionesImpares ( ) {

```
}
```

Imprimir el dato de la mitad de la cadena.

```
public void imprimirElementoMitad ( ) {



}
```

Imprimir los caracteres que hay en la cadena de la última posición a la primera.

```
public void imprimirUltimoElemento ( ) {



}
```

Imprimir todos los caracteres de la cadena (sugerencia: tenga en cuenta que este método debe incluir un ciclo).

```
public void imprimirCadena ( ) {



}
```

7.9 Referencias bibliográficas

Aguilar, L. J. (2004). *Fundamentos de programación algoritmos y estructura de datos*. Tercera edición. Ciudad: México. McGraw-Hill.

Barnes, D. y Kölling, M. (2007). *Programación orientada a objetos con Java*. Ciudad: Madrid. Prentice Hall.

Bertrand, M. (1999). *Construcción de software orientado a objetos*. Segunda edición. Ciudad: Madrid. Prentice Hall.

Booch, G. (1996). *Análisis y diseño orientado a objetos con aplicaciones*. Ciudad: México. Addison Wesley.

Deitel y Deitel (2012). *Cómo programar en Java*. Ciudad: México. Editorial Pearson (Prentice Hall).

Fowler, M. y Scott, K. (1999). *UML gota a gota*. Ciudad: México. Editorial Pearson.

Graig, L. (2003). *UML y patrones: una introducción al análisis y diseño orientado a objetos y al proceso unificado*. Segunda edición. Ciudad: México. Prentice Hall.

Rumbaugh, J., Jacobson, I. y Booch, G. (2005). *The Unified Modeling Language User Guide*. Ciudad: San Francisco. Addison-Wesley, Reading, Mass. ISBN-13: 078-5342267976, ISBN-10: 0321267974.

Villalobos, J. y Casallas, R. (2006). *Fundamentos de programación, aprendizaje activo basado en casos*. Ciudad: México. Editorial Pearson.

Capítulo 8

Relaciones entre clases

8.1 Temática a desarrollar

- Cardinalidad
- Asociación
- Agregación
- Composición
- Generalización
- Dependencia

8.2 Introducción

En una aplicación de software, lo más normal es que se involucren dos o más clases; estas clases de alguna manera deben relacionarse entre sí. En la construcción de un proyecto de software, están presentes los diagramas de clases, los cuales representan la estructura estática de un sistema en términos de clases y de relaciones entre estas.

En el modelado orientado a objetos, las relaciones entre clases más importante son:

- Asociación: es una conexión entre clases que implica la existencia de una relación estructural entre objetos de esas clases.
- Generalización: es una relación entre una clase más general y una más específica o especializada.
- Dependencia: es una relación de uso entre clases.

El mapa mental de la figura 22 describe un poco más pormenorizado el concepto de relación entre clases.

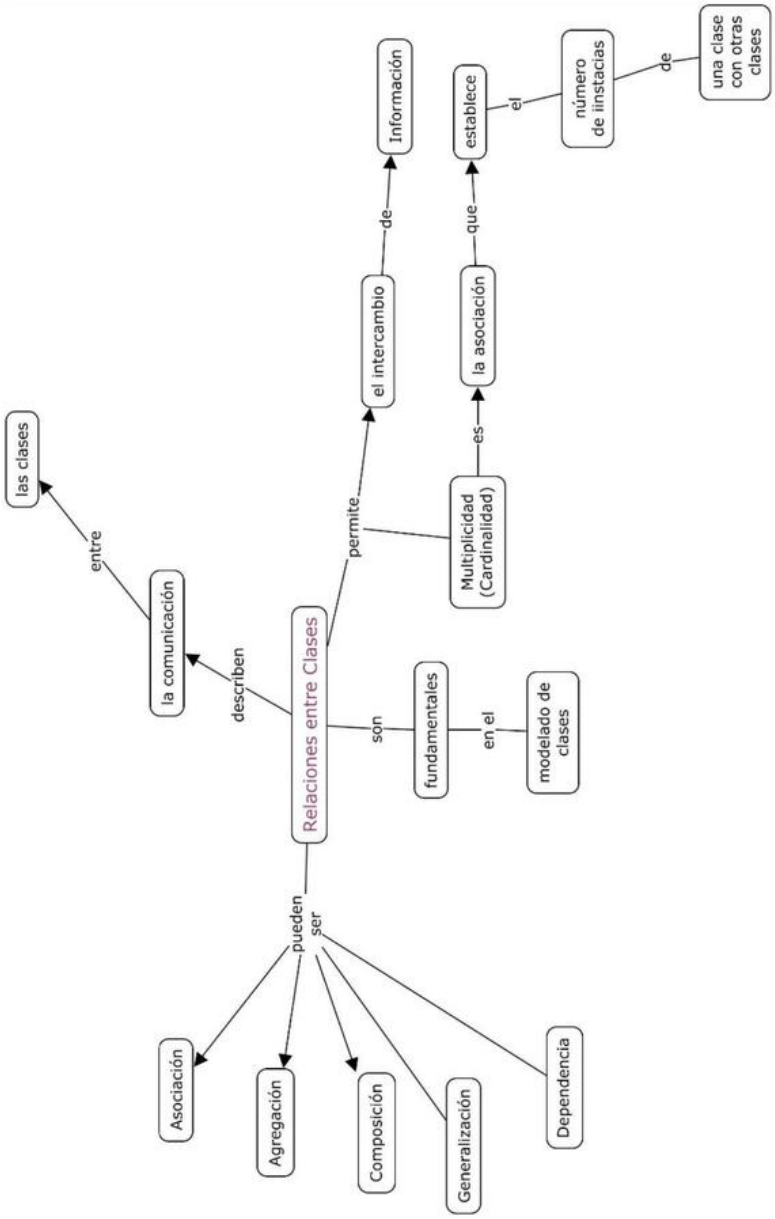


Figura 22.
Mapa mental de relaciones entre clases

Este capítulo es una introducción del tema de modelado de clases que serán aplicado en las temáticas de los capítulos subsiguientes.

8.3 Elementos entre las relaciones de clases

8.3.1 Cardinalidad

La cardinalidad hace referencia a la cantidad de objetos que participan en la relación determinada. Existen tres tipos de multiplicidad o cardinalidad, a saber:

- Uno a uno.
- Uno a muchos.
- Muchos a muchos.

La tabla 14 muestra las diferentes notaciones para la lectura de los tipos de cardinalidad.

Tabla 14.
Tipos de cardinalidad

Notación	Lectura
1	Exactamente uno
*	Muchos
0…1	Cero a uno
0…*	Cero a muchos
1…*	Uno a muchos (al menos uno)
M…N	De M hasta N (enteros naturales)

La cardinalidad es una restricción a una asociación que limita el número de objetos relacionados de una clase, como, por ejemplo:

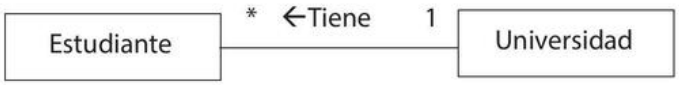
- Con un número fijo: 1.
- Con un intervalo de valores: 2..5.
- Con un rango en el cual uno de los extremos es un asterisco. Significa que es un intervalo abierto. Por ejemplo: 2...* significa 2 o más.
- Con una combinación de elementos como los anteriores separados por comas: 1, 3..5, 7, 15..*.

- Con un asterisco (*). En este caso indica que puede tomar cualquier valor (cero o más).

Ejemplos:

1. La universidad tiene un rector y el rector dirige la universidad (relación uno a uno).

2. La universidad tiene muchos estudiantes.



8.3.2 Asociación

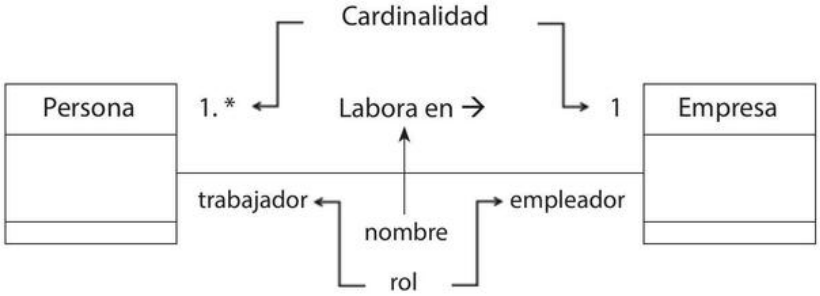
Una asociación representa una relación entre clases y es el mecanismo que permite a los objetos comunicarse entre sí. Describe la conexión entre diferentes clases (la conexión entre los objetos reales se denomina conexión de objetos o enlace).

Las asociaciones pueden ser unidireccionales o bidireccionales (indicando si los dos objetos participantes en la relación pueden intercambiar mensajes entre sí o es únicamente uno de ellos el que recibe información del otro). Cada extremo de la asociación también tiene un valor de cardinalidad, que indica cuántos objetos de ese lado de la asociación están relacionados con un objeto del extremo contrario.

En UML, las asociaciones se representan por medio de líneas que conectan las clases participantes en la relación y también pueden mostrar el papel y la multiplicidad de cada uno de los participantes. La multiplicidad se muestra como un rango [mínimo-máximo] de valores no negativos, con un asterisco (*) representando el infinito en el lado máximo.

Las asociaciones tienen algunos elementos o atributos que están presentes, como lo son:

- Nombre: describe de una forma más específica la naturaleza de la relación entre los objetos. No suele ser necesario, salvo que haya varias asociaciones, por ejemplo, "trabaja en" o "dirige" para una asociación entre empleados y departamentos.
- Roles: define el nombre de un objeto desde el otro. Por ejemplo, trabajador y empleador para una relación entre varias personas y empresa.
- Cardinalidad: permite definir el número de objetos de ambos tipos. Por ejemplo, en una empresa laboran muchos trabajadores.



En programación orientada a objetos, el comportamiento de un sistema se define en términos de interacciones entre objetos, es decir, intercambios de mensajes. Enviar un mensaje habitualmente resulta en la invocación de una operación en el receptor. Las asociaciones son necesarias para la comunicación, ya que los mensajes son enviados a través de las asociaciones; sin asociaciones, los objetos quedarían aislados e incapaces de interactuar.

Ejemplo:

Un cliente tiene una cuenta de ahorros. Asociación bidireccional con cardinalidad 1-1.

```
public class Cliente {
  private String nombre;
  private String apellido;
  private CuentaAhorros unaCuenta-
DeAhorros;

}
```

```
public class CuentaAhorros {
        private String
numeroCuenta;
        private double saldo;
        private Cliente unCliente;
}
```

2. Asociación direccional con cardinalidad 1-1.

Relación direccional que indica que el usuario tiene un *login*. Conformado por un login y una clave.

```
public class Usuario {
private String nombre;
private String apellido;
private Login unLogin;

}
```

```
public class Login {
        private String login;
        private String clave;

}
```

8.3.3 Agregación

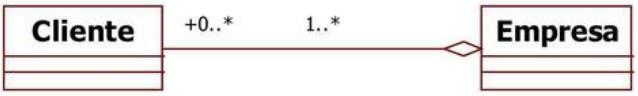
Es una relación del tipo todo-partes, donde los objetos de una clase están compuestos por objetos de otra clase. Por ejemplo, un automóvil está compuesto por motor, ruedas, puertas, entre otros.

La representación en UML del símbolo de agregación es un diamante colocado en el extremo en el que está la clase que representa el "todo".

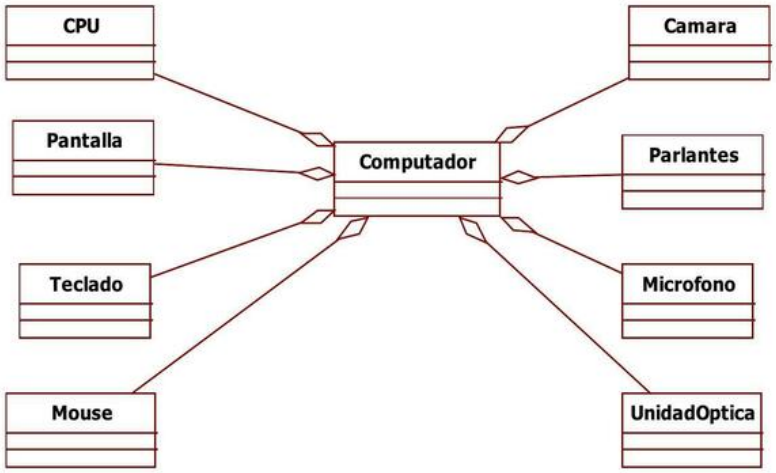
La clase B está conformada por objetos de tipo A (uno-muchos).

Ejemplos:

- Un automóvil tiene llantas.
- Un teléfono celular tiene un *display* o pantalla.
- Se tienen las clases Cliente y Empresa donde una empresa reúne a varios clientes.



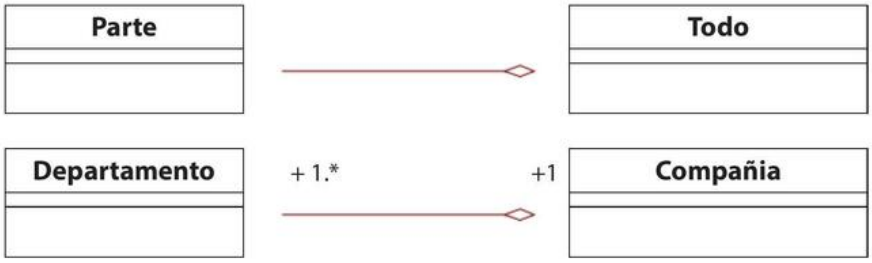
- Un computador es el resultado de una agregación, ya que está integrado por una serie de dispositivos, como la Unidad Central de Proceso (CPU), un monitor o pantalla, un teclado, un *mouse*, una cámara, dos parlantes, un micrófono, una unidad óptica (CD, DVD, Blu-ray), entre otros. En otras palabras, se puede aplicar una prueba a un objeto de la clase Computador ya que está conformado o tiene una CPU, un teclado, un *mouse*, una cámara, dos parlantes, un micrófono, una unidad óptica. El modelo de clases sería el siguiente:



8.3.4 La composición

La composición es un tipo especial de agregación que denota una fuerte dependencia de la clase Todo a la clase Parte. Se grafica con una flecha rombo de diamante relleno, desde la *parte* hacia el *todo*.

Ejemplo:



La composición se caracteriza por:

- Es dependencia fuerte, ya que el elemento dependiente desaparece al destruirse el que lo contiene y, si es de cardinalidad 1, es creado al mismo tiempo.

- Hay una pertenencia fuerte. Se puede decir que el objeto contenido es parte constructiva y vital del que lo contiene.

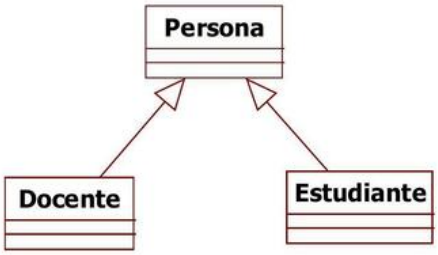
8.4 La relación de especialización/generalización

Esta relación indica que una clase (clase derivada) hereda los métodos y atributos especificados por una clase (clase base), por lo cual una clase derivada, además de tener sus propios métodos y atributos, podrá acceder a las características y atributos visibles de su clase base (*public* y *protected*).

Características:

- La generalización es la relación entre una clase (superclase) y una o más versiones especializadas (subclases).
- Los atributos y métodos comunes a un grupo de subclases se asocian a la superclase y son compartidos por las subclases.
- Se dice que cada subclase hereda las características de la superclase, aunque también puede agregar, es decir, incluir sus propios atributos y operaciones específicas.
- La notación UML para la generalización es una flecha que conecta la superclase con sus subclases, apuntando a la superclase.
- Junto a la punta de flecha en el diagrama, pueden colocarse un discriminador, un atributo de tipo enumeración que indica la propiedad de un objeto que se abstrae en una relación de generalización particular (visualizado en el ejemplo).

Ejemplo:



En este diseño de clases se especifica que las clases Estudiante y Docente heredan de la clase Persona, es decir, Estudiante y Docente podrán acceder a las características de Persona. También puede tener su respectiva diferenciación, ya que un estudiante puede obtener sus notas previa evaluación realizada por parte de un profesor. Otra forma de identificar la relación desde la lectura es: todo estudiante es una persona y todo docente es una persona.

8.5 Dependencia

Cuando los objetos de una clase utilizan objetos de otra clase, existe una relación de dependencia entre las respectivas clases.

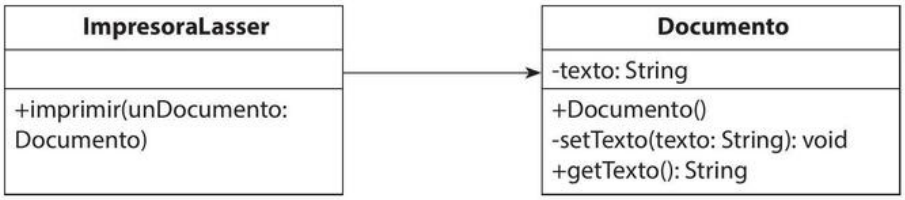
En UML, esta relación se representa en el diagrama de clases con una flecha y un trazo discontinuo en el sentido de las clases cuyos objetos usan los de otra clase; esto depende de la especificación de la clase usada. Si cambia la especificación, habrá que hacer cambios en las clases que la usan.

Del diagrama de clases se puede observar que:

- La ClaseA usa a la ClaseB.
- La ClaseA depende de la ClaseB.
- Dada la dependencia, todo cambio en la ClaseB podrá afectar a la ClaseA.
- La ClaseA conoce la existencia de la ClaseB, pero la ClaseB desconoce que existe la ClaseA.

Las dependencias generalmente se utilizan para indicar que una clase utiliza a otra como argumento en alguna operación o sus objetos utilizan alguna de las operaciones de la otra clase.

Se tiene la clase Documento, la cual incluye al atributo texto y la clase ImpresoraLasser se encarga de imprimir los documentos:



8.6 Lecturas recomendadas

- Grado de la asociación.
- Técnicas comunes del modelamiento.

8.7 Preguntas de evaluación

- ¿Qué es el modelado de colaboraciones simples?
- ¿Qué es la ingeniería directa?
- ¿Qué es la ingeniería inversa?

8.8 Ejercicios

Dependencia

1. Completar la información de la siguiente tabla.

Descripción	Notación
Un vuelo necesita una reservación.	
Un vuelo tiene muchas reservaciones.	
Un vuelo tiene un horario.	
Un vuelo tiene un origen.	
Un vuelo pertenece a una aerolínea.	
Un pasajero compra un tiquete.	
Un pasajero compra muchos tiquetes.	
Un pasajero tiene un pasaporte.	
Un equipo de fútbol tiene muchos jugadores.	
Una persona tiene una cédula.	
Una empresa tiene cero o más empleados.	

2. Plantear tres ejemplos para cada uno de los siguientes enunciados y los correspondientes diagramas de clases.

   - Número fijo: m (m denota el número).
   - Describiendo una multiplicidad de "uno-uno".
   - Describiendo una multiplicidad de "uno-muchos".

Agregación

1. Completar la información de la siguiente tabla que resume algunas diferencias entre agregación y composición.

Característica	Agregación	Composición
Representación	Rombo sin color	Rombo relleno
Puede tener varias asociaciones que comparten los componentes	Sí	
Destrucción de los componentes al destruir el compuesto		
Cardinalidad a nivel del compuesto		

2. Construir un diagrama de clases en donde se identifique que un auto está integrado por ruedas, caja de cambios, puerta y motor.
3. Modelar en un diagrama de clases un computador conformado por la CPU, el teclado, el monitor y el *mouse*.
4. Construir una relación de clases que permita modelar la relación dada por la frase "la universidad tiene varias facultades".
5. Elaborar el diagrama de clases de la relación que existe entre un texto o libro y sus páginas.

Asociación

1. Construir el diagrama de clases de la asociación juego-participante, donde el juego:
   - Tiene dos participantes.
   - Tiene cuatro participantes.
2. Construir el diagrama de clases y el código en Java para una relación en donde un cliente puede tener asociadas muchas órdenes de compra y una orden de compra solo puede tener asociado un cliente.

3. Modelar la relación empresa-empleado a partir de diagramas de clases donde un empleado puede trabajar en varias empresas (pluriempleo) y, por supuesto, que una empresa puede estar formada por más de un empleado.
4. Consultar los tipos de asociación que existen y su navegabilidad.
5. Elaborar el diagrama de clases donde un gerente dirige un departamento, un departamento tiene varios empleados y varios empleados trabajan en varios proyectos.

Agregación

1. Construir el modelo de clases en donde se tienen las clases Agenda y Contacto, donde una agenda agrupa varios contactos.
2. Elaborar un diagrama de clases que modele la relación entre un avión y sus dos alas.
3. Diseñe un diagrama de clases que permita modelar las relaciones dadas entre la universidad, cursos, estudiantes y maestros que describen una universidad con muchos estudiantes, estos asisten a diferentes cursos y son orientados por diferentes maestros.

Generalización

1. Se requiere construir el modelo de diagrama de clases para el siguiente enunciado: una colección de autos, conformada por autos deportivos, ambulancias, camiones. Todas estas clases podrían derivar de una clase Automóvil, de manera que se aprovechase la definición común de parte de la implementación. Cada clase añadiría los detalles relativos a esa clase particular. Así, la clase Ambulancia añadiría el método encenderSirena() o la clase Camión podría tener la propiedad booleana estaCargado.

2. Una compañía editorial produce una serie de libros impresos como audiolibros en discos compactos. Se requiere construir el modelo de clases conformado por:
   - Una clase denominada Publicación que registra el título del libro y el precio de una publicación. A partir de esta clase, derive dos clases:
     - La clase Libro, a la cual le agregue el número de páginas (número entero) y
     - La clase CD, a la cual le agregue el tiempo de reproducción en minutos (numérico real).

Cada una de las clases debe tener propiedades para acceder a sus respectivos datos. Elabore un diagrama de clases UML indicando las relaciones de herencia

y codifique un sistema mediante el cual se generen instancias de las clases Libro y CD, donde el usuario capture sus datos y se inserten en los respectivos objetos.

Dependencia

1. Diseñar el diagrama de clases que modele la relación de las entidades Curso y PlanificacionCurso, donde la clase PlanificaciónCurso implementa los métodos adicionarCurso y cancelarCurso, los cuales reciben como parámetro un objeto de la clase Curso y se aplica el concepto de dependencia.
2. Codificar en Java el anterior modelo de clases.

8.9 Bibliografía

Weitzenfeld, A. (2005). *Ingeniería de software orientada a objetos con UML, Java en Internet*. Ciudad: México. Thomson.

Booch, G. (1996). *Análisis y diseño orientado a objetos con aplicaciones*. Ciudad: México. Addison Wesley.

Capítulo 9

Abstracción y encapsulamiento

9.1 Temática a desarrollar

- Abstracción
- Encapsulamiento

9.2 Introducción

La Programación Orientada a Objetos (POO) es un paradigma de programación que usa objetos y sus interacciones para el diseño y construcción de aplicaciones de software por computador. Es un paradigma, ya que existen metodologías de soporte (como OpenUP, Scrum, XP), elementos que lo representan (como los objetos) y herramientas tecnológicas que lo soportan (como C++, Java, C#, entre otras).

Los pilares de la programación orientada a objetos son abstracción, encapsulamiento y polimorfismo. Son los elementos más importantes que deben tener los objetos de software para cumplir con el paradigma de orientación a objetos.

9.3 Abstracción

La abstracción se puede considerar como la capacidad de integrar las propiedades y comportamientos necesarios para la correcta representación del objeto dentro del sistema. Es decir, no se tienen en cuenta aspectos del sistema en estudio que no son de interés, con el objeto de concentrarse en aquellos que son relevantes para modelar el sistema.

Dentro de las propiedades o características esenciales, se encuentran los atributos (o datos) y los comportamientos que corresponden a los métodos. Al modelar, se piensa en objetos y es necesario tomar las características y propiedades de un elemento real y llevarlo a un objeto. La abstracción es la base fundamental para el diseño de las aplicaciones de software.

Java proporciona un amplio conjunto de clases útiles para desarrollar aplicaciones, como es el caso de las colecciones, que se encuentran definidas en el paquete java.util y las cuales representan un grupo de objetos que se pueden emplear en diferentes aplicaciones. Para utilizarlas en un programa fuente Java, se deben importar haciendo uso de import.

Para las estructuras de datos, el programador puede contar con una serie de recursos de software y, en tal caso, se tiene el tipo de dato abstracto (TAD), el TAD Pila, el TAD Cola y otras estructuras de uso común en una aplicación con manejo dinámico de memoria.

Las API de Java brindan un conjunto de clases útiles para realizar toda clase de tareas necesarias dentro de un programa. La API proporciona una interfaz de Java con un conjunto de métodos y clases de utilidad que se puede utilizar para escribir aplicaciones en Java (java.lang, java.io, java.sql); esto es otra forma de abstracción.

La abstracción es un principio de la programación orientada a objetos y es una herramienta que permite tratar la complejidad y el entorno del problema. En general, una aplicación de software es una descripción abstracta de una actividad o procedimiento que sucede en el mundo real.

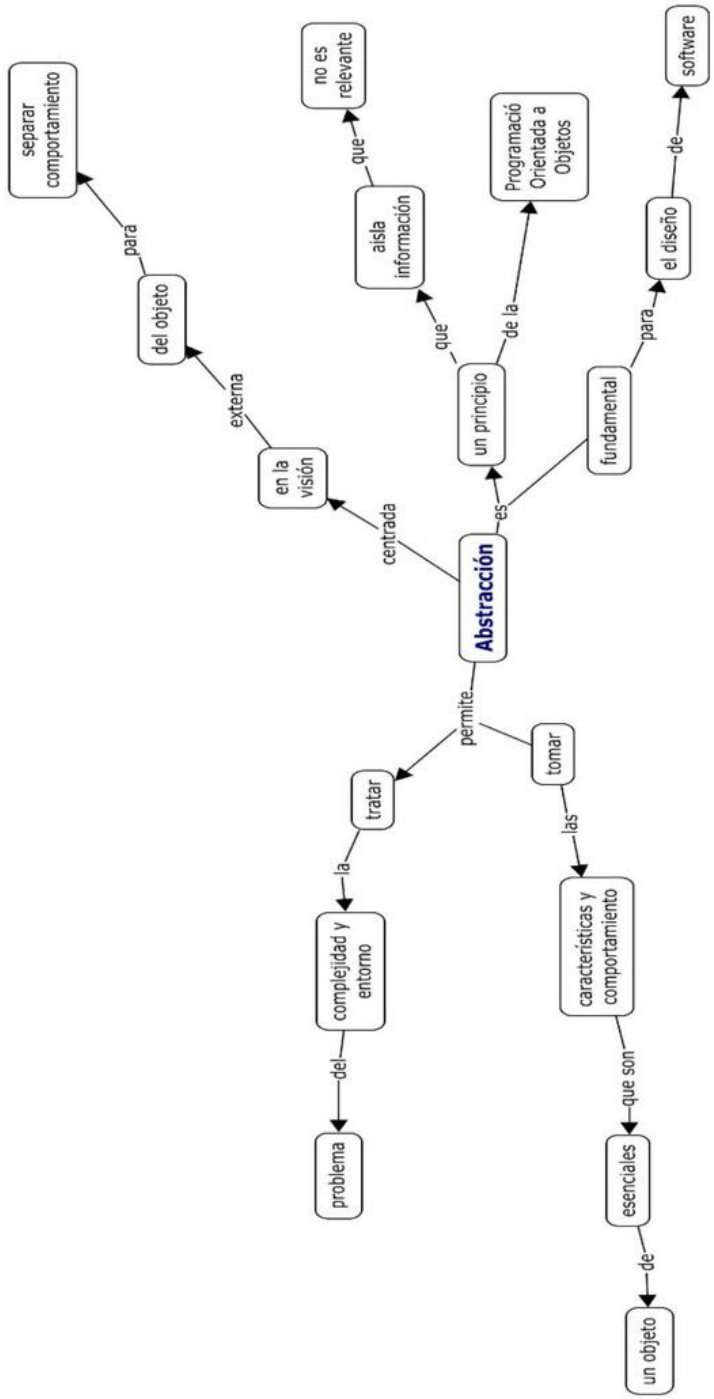
Ejemplos:

- Las diferentes escalas en mapas.
- Un automóvil es una entidad de la que se identifican características como el color, la marca, el modelo, el precio, la placa, entre otras. Y algunos de los métodos o funciones típicas de esta entidad son: girar a la derecha, girar a la izquierda, frenar, encender, retroceder, avanzar.

La gerencia de un parqueadero requiere un sistema para controlar los vehículos que ingresan a sus instalaciones. En este caso, dentro de las características de la clase Vehículo están: la marca, el modelo, el propietario: número de documento, nombres y apellidos, dirección, teléfono y correo.

La figura 23 del mapa mental de abstracción complementa este principio de la programación orientada a objetos.

Figura 23.
Mapa mental de abstracción



9.4 Encapsulamiento

El encapsulamiento significa reunir a todos los elementos bajo un mismo nombre, esto es, atributos y operaciones que pueden considerarse pertenecientes a una misma entidad.

El encapsulamiento corresponde a la posibilidad que tiene un objeto de ocultar los datos y los métodos que le son propios y que serán solo accesibles dentro del propio objeto.

La abstracción y el encapsulamiento están relacionados, ya que la abstracción propende por determinar las características y comportamientos de un objeto y el encapsulamiento se ocupa de la implementación establecida por el comportamiento del objeto.

En Java, el encapsulamiento garantiza que los usuarios de un objeto no puedan modificar el estado del objeto sin usar una interfaz (correspondiente al conjunto de métodos que son accesibles a otros objetos). Una ventaja del encapsulamiento es que, si un módulo cambia internamente sin modificar su interfaz, el cambio no supondrá ninguna modificación del sistema.

El encapsulamiento en un sistema orientado a objetos se representa en cada clase u objeto definiendo sus atributos y métodos con los siguientes modos de acceso:

- **public:** representado con el signo más (+), tanto para atributos como para métodos, los cuales son accesibles dentro y fuera de la clase, pudiendo ser llamados por cualquier clase.
- **private:** representado con el signo menos (-), ya sean atributos o métodos, los cuales solo son accesibles dentro de la misma clase.
- **protected:** representado con el signo numeral (#) para los atributos o los métodos que son accesibles dentro de la implementación de la misma clase y sus clases derivadas o subclases (clases hijas).
- **default:** sucede cuando no se especifica ningún modificador de acceso y entonces se utiliza el nivel de acceso por defecto, que consiste en que el elemento puede ser accedido solo desde las clases que pertenezcan al mismo paquete.

La figura 24 del mapa mental de encapsulamiento complementa este principio de la programación orientada a objetos.

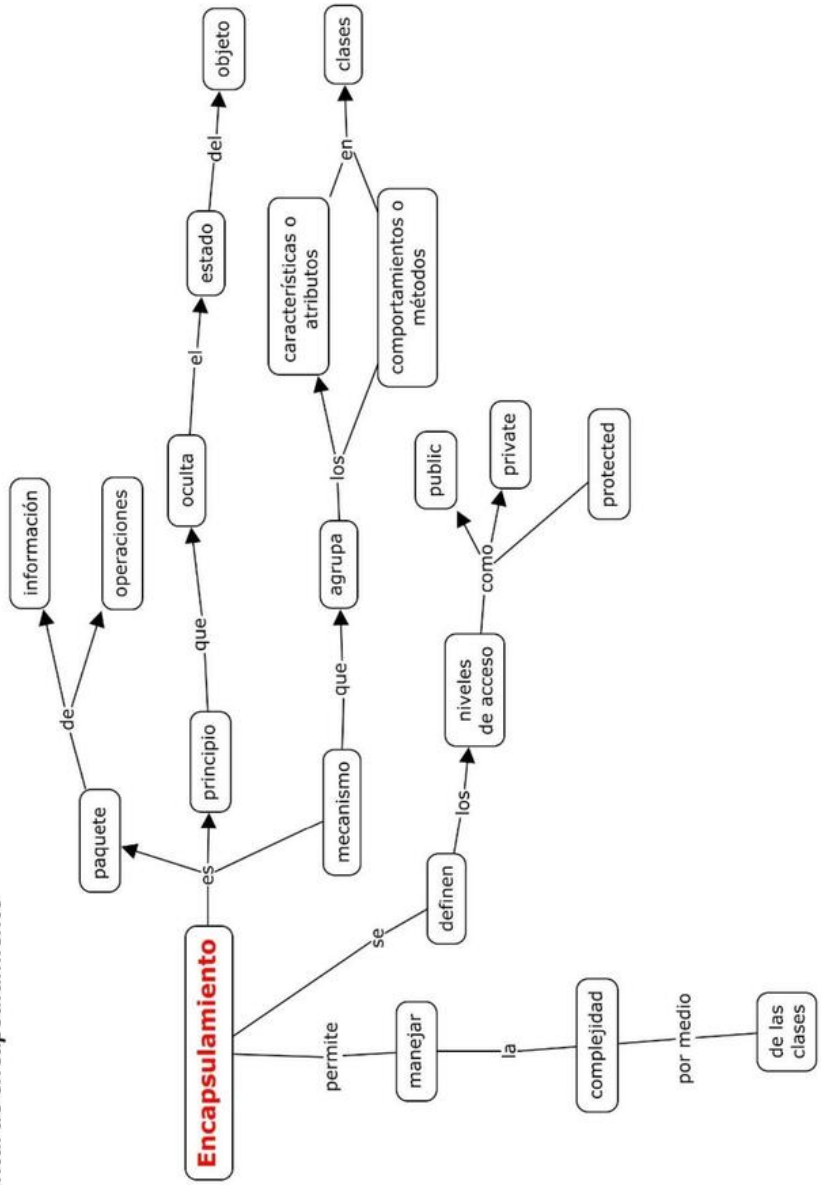


Figura 24.
Mapa mental de encapsulamiento

Ejemplos:

1. El reproductor de DVD muestra las acciones que se pueden realizar en el tablero de mando o en el control remoto, por ejemplo, *play*, *pause*, *stop*, *review*, *zoom*, entre otras; el usuario utiliza el DVD por lo que ofrece y no ingresa en el interior para hacer nuevas acciones o realizar alguna de ellas.

2. La clase Persona encapsula los atributos del nombre, la edad, el método constructor y los métodos set y get.

```
class Persona {
        private String nombre;
        private int edad;

        public Persona(String nombre, int edad) {

                super();
                this.nombre = nombre;
                this.edad = edad;
        }

        public String getNombre() {
                return nombre;
        }

        public void setNombre(String nombre) {
                this.nombre = nombre;
        }

        public int getEdad() {
                return edad;
        }


        public void setEdad(int edad) {
                this.edad = edad;
        }
}
```

Persona
-nombre: String -edad: int
+Persona(nombre: String, edad: int) +getNombre(): String +setNombre(nombre: String) +getEdad(): int +setEdad(edad: int)

La tabla 15 establece un comparativo entre los conceptos de encapsulamiento y abstracción.

Tabla 15.
Comparación entre abstracción y encapsulamiento

Abstracción	Encapsulamiento
La abstracción resalta los aspectos importantes de un objeto.	Exige que sus características y métodos estén bien definidos y no se confundan con los de otros.
No se incluyen detalles sobre la implementación de las operaciones.	Oculta los detalles internos de un objeto.
A través de la abstracción conseguimos extraer las cualidades principales sin detenernos en los detalles.	Especifica los datos y comportamiento lógico que está oculto en un objeto.
Es una simplificación o modelo de un concepto complejo, proceso o un objeto del mundo real.	Permite la visibilidad de atributos y/o métodos, en una determinada clase, a partir de los modificadores de acceso *private*, *protected* y *public*.
Permite simplificar la comprensión del mundo del problema. Es la forma general de ver a un objeto, sin conocer su composición interior u otros componentes.	La encapsulación en programación significa agrupar a todos los componentes de un objeto en uno solo.
Permite resaltar las características y comportamientos relevantes de un objeto.	Corresponde a agrupar a todos los componentes de un objeto en uno solo.

9.5 Lecturas recomendadas

- Paradigma de programación orientada a objetos.
- Principios de la programación orientada a objetos.

9.6 Preguntas de revisión de conceptos

- ¿Cómo identificar que se debe aplicar abstracción para resolver un problema?

- ¿Cómo identificar que se debe aplicar encapsulamiento para resolver un problema?

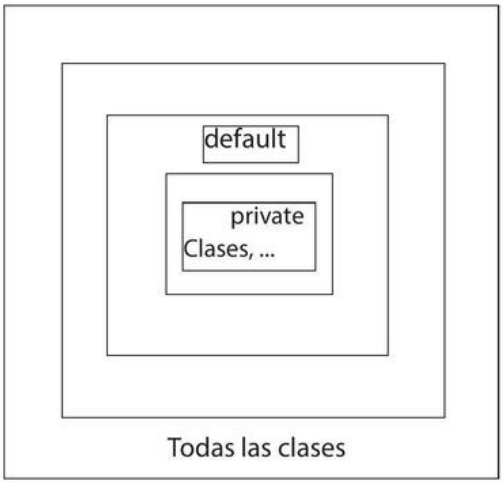
9.7 Ejercicios

Abstracción

1. Determinar las características que se pueden abstraer de los estudiantes de la universidad.
2. Obtener las características de los computadores del laboratorio y sus comportamientos.
3. Obtener las características de los pupitres de un salón y sus comportamientos.
4. Liste siete elementos que usted emplea normalmente sin conocer cómo es el funcionamiento interno.

Encapsulamiento

1. Identificar y nombrar las características y operaciones de un celular.
2. Reconocer y nombrar las características y operaciones de un libro.
3. Modelar los atributos y operaciones de un computador.
4. Completar la información de la siguiente gráfica con base a los modificadores de acceso y qué elementos involucran:



- Completar la información de la siguiente tabla.

Modificadores de acceso				
	La misma clase	**Otra clase del mismo paquete**	**Subclase de otro paquete**	**Otra clase de otro paquete**
public				
protected				No
default			No	
private		No		

9.8 Bibliografía

Barnes, D. y Kölling, M. (2007). *Programación orientada a objetos con Java*. Ciudad: Madrid. Prentice Hall.

Booch, G. (1996). *Análisis y diseño orientado a objetos con aplicaciones*. Ciudad: México. Addison Wesley.

Deitel y Deitel (2012). *Cómo programar en Java*. Ciudad: México. Editorial Pearson (Prentice Hall).

Fowler, M. y Scott, K. (1999). *UML gota a gota*. Ciudad: México. Editorial Pearson.

Villalobos, J. y Casallas, R. (2006). *Fundamentos de programación, aprendizaje activo basado en casos*. Ciudad: México. Editorial Pearson.

Capítulo 10

Herencia

10.1 Temática a desarrollar

- Herencia
- Superclase
- Subclase
- Protected
- Extends
- Super

10.2 Introducción

La Programación Orientada a Objetos (POO) es una forma especial de programar, más cercana a cómo se expresan o representan las cosas en la vida real que otros tipos de programación; es un paradigma que integra un conjunto de teorías, estándares, modelos y métodos que permiten modelar el dominio de un problema o proyecto e implementarlo en un lenguaje de programación para brindar la solución a ese requerimiento.

El paradigma orientado a objetos tiene unas características que lo definen y lo diferencian de los demás paradigmas, como lo son: abstracción, encapsulamiento, modularidad, herencia y polimorfismo, entre otras. En este capítulo, se aborda el concepto de herencia, donde la idea básica es poder crear clases basadas en clases ya existentes. Cuando se hereda de una clase existente, se está reutilizando código (métodos y atributos). En otras palabras, la herencia permite que los sistemas orientados a objetos puedan definir clases en términos de otras clases.

En el mapa mental de la figura 25 se complementa el concepto de herencia.

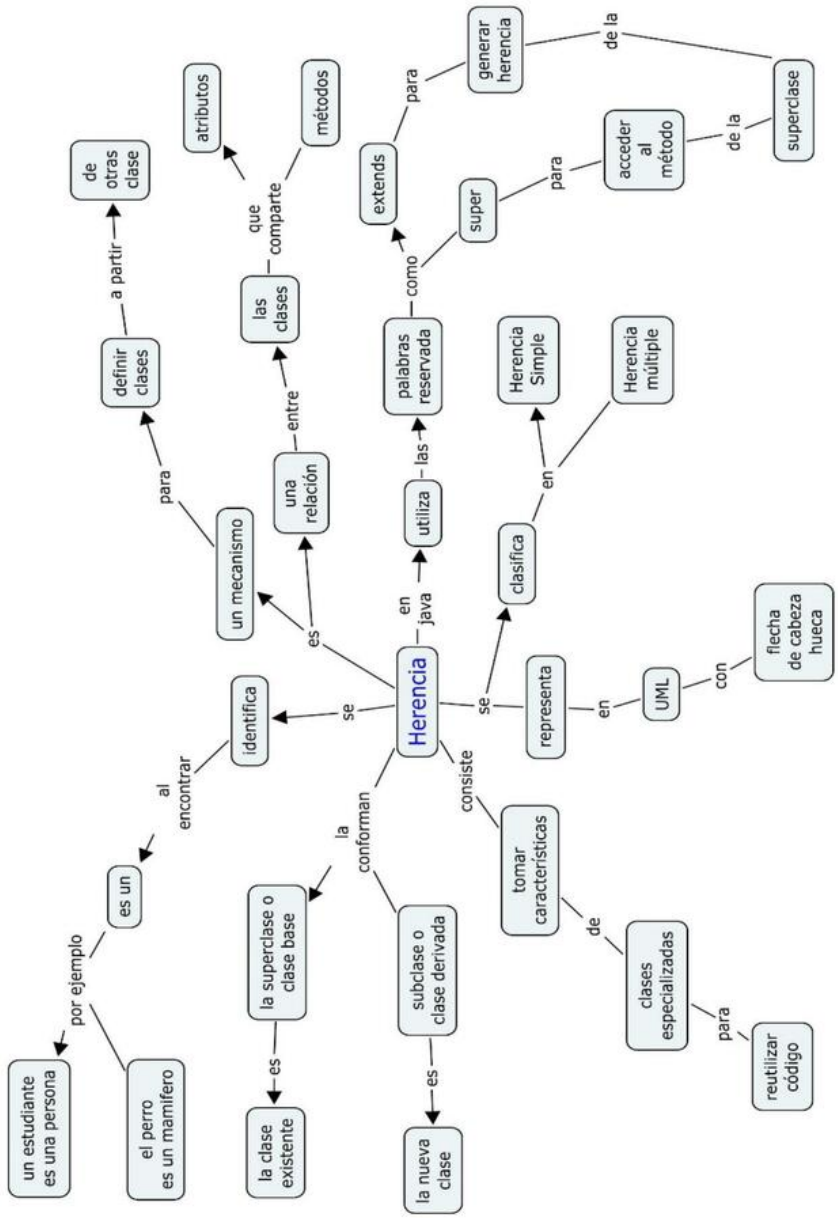


Figura 25.
Mapa mental de herencia

10.3 Herencia

Es uno de los pilares fundamentales de la programación orientada a objetos; es un mecanismo que permite definir nuevas clases a partir de otras ya definidas, la idea básica es poder crear clases basadas en clases ya existentes, de forma que heredan las propiedades de la clase base.

La herencia permite implementar las relaciones en las cuales un objeto pertenece a una clase que es hija de otra clase, lo que significa que la clase hija hereda, de su clase padre, la estructura y el comportamiento de esta.

Ejemplos:

Animales mamíferos

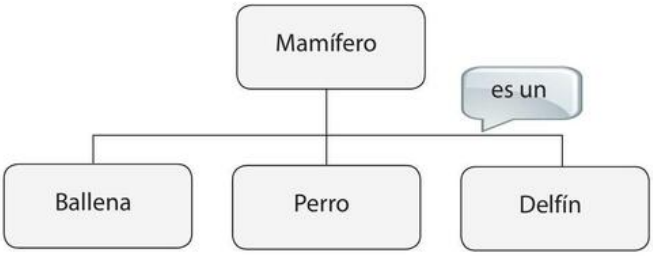
Características comunes

Ballena	Perro	Delfín
color tamaño peso edad	color tamaño peso edad	color tamaño peso edad

Las características comunes y propias de cada uno pueden ser el color, el tamaño, el peso y la edad y se pueden agrupar en una entidad que puede ser Mamífero.

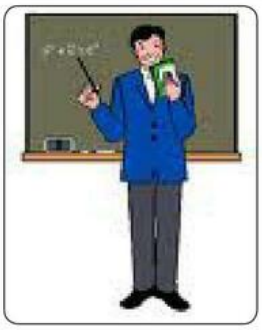
Una ballena ***es un*** Mamífero.
Un perro ***en un*** Mamífero.
Un delfín ***es un*** Mamífero.

Un perro, un delfín y una ballena son objetos diferentes, pero tienen una cosa en común: los tres son mamíferos. Así como lo son los delfines y las ballenas, aunque se muevan en un entorno diferente, entonces un perro es un mamífero, un mamífero es un animal, un animal es un ser vivo, entre otros. Lo cual permite establecer la relación *es un*.



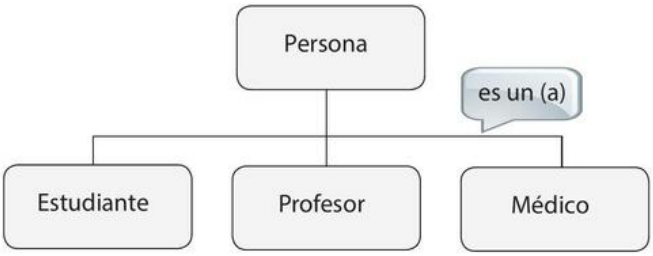
Ejemplo:

Diferentes personas
Características comunes

Estudiante	 Profesor	Médico
nombre apellido edad sexo	nombre apellido edad sexo	nombre apellido edad sexo

Se puede observar que sí hay características comunes y propias de cada uno (estudiante, profesor y médico). Los atributos comunes permiten agrupar en una sola entidad denominada Persona.

- Un estudiante ***es una*** Persona.
- Un profesor ***es una*** Persona.
- Un médico ***es una*** Persona.



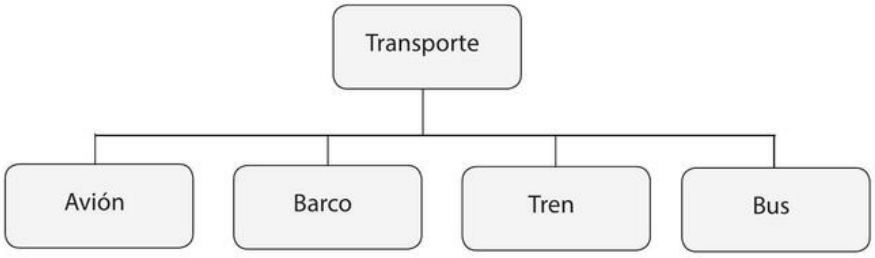
La herencia propende por utilizar una clase ya creada, como la clase Mamífero, para tomar las características (color, tamaño, peso, edad) en clases más especializadas o derivadas, para el caso, perro, ballena, delfín, y a partir de la clase base reutilizar el código que sea común con la clase base y solamente definir nuevos métodos o redefinir algunos de los existentes.

Ejemplo:

Medios de transporte

Bus	Tren	Avión	Barco
marca modelo color velocidad	marca modelo color velocidad	marca modelo color velocidad	marca modelo color velocidad

- Avión ***es un*** Transporte
- Barco ***es un*** Transporte
- Bus ***es un*** Transporte
- Tren ***es un*** Transporte



Cuando se declara una clase, se definen las características o variables y los métodos o comportamientos que pueden tener los objetos que pueden ser creados a partir de la clase.

Por ejemplo, a partir de la clase Persona, se define nombre, apellido y el género, con sus respectivos constructores y métodos de acceso y modificación de las variables de instancia.

Persona
-nombre: String -apellido: String -genero: char +Persona() +Persona(nombre: String, apellido: String, genero: char) +getNombre(): String +setNombre(nombre: String) +getApellido(): String +setApellido(apellido: String) +getGenero(): char +setGenero(genero: char) +toString(): String

Al definir la clase Estudiante, el cual tiene un nombre, un apellido y un género, no es necesario empezar desde cero; se puede decir que la clase Estudiante sería una clase que hereda de la clase Persona. Solamente a la clase Estudiante se anexarían el atributo matrícula y los métodos como los constructores y los métodos de acceso y modificación que sean solo específicos al estudiante. Esto significa que, cuando se hereda de una clase existente, además de reutilizar

atributos, también se pueden agregar nuevos atributos y métodos para cumplir con la nueva clase.

Los atributos comunes a todas las clases Persona y Estudiante se definen en la clase base. Los métodos comunes en la clase padre y el atributo matrícula en la clase Estudiante, que es la clase hija.

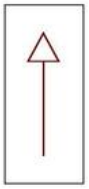
Estudiante
-matricula: long
+Estudiante() +Estudiante(matricula: long) +setMatricula(matricula: long) +getMatricula(): long +toString(): String

Las relaciones de herencia se establecen entre clases, puesto que cubren a todos los objetos que se crearán como instancias de esas clases. La clase padre se denomina superclase, es decir, la clase de la cual heredan o se derivan otras clases, y las clases hijas, o las que heredan, se denominan subclases.

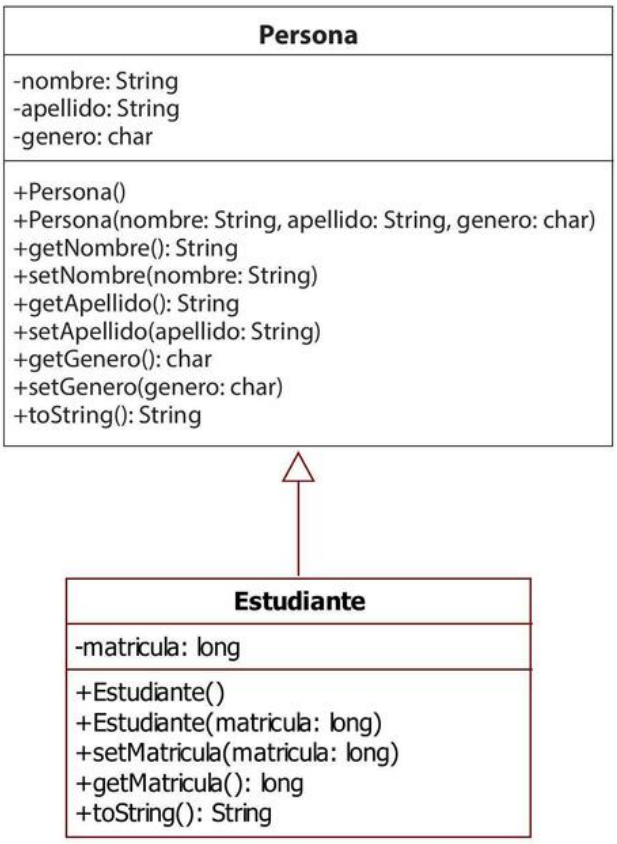
Para el ejemplo, la clase Persona es la superclase, la cual agrupa las clases con propiedades y comportamientos comunes, y la clase Estudiante es la clase derivada o clase hija.

10.4 Representación de la herencia

En diagramas UML de clases se hace con una flecha de cabeza hueca.



El diagrama de clases sería el siguiente:



Java solo maneja herencia simple; en el ejemplo, la subclase Estudiante comparte la estructura y el comportamiento de una sola superclase Persona, en otras palabras, una subclase puede heredar datos y métodos de una única clase.

El símbolo # que se antecede a los atributos nombre, apellido y género corresponde al modificador protected, que puede ser utilizado para un atributo o un método, indicando que es accesible en la clase en donde se define y en cualquiera de sus subclases, en otras palabras, al heredar sí se puede usar desde la clase derivada. Para indicar que una clase deriva de otra heredando sus propiedades (métodos y atributos), en Java se usa la palabra extends. Codificando las clases del ejemplo sería:

La superclase, clase base o clase padre:

```
public class Persona {
protected String nombre;
        ....
}
```

La subclase, clase derivada o clase hija:

```
public class Estudiante extends Persona{
        private long matricula;
  .....

}
```

La palabra reservada extends permite en Java definir la herencia, entonces, con la sentencia public class Estudiante extends Persona se está diciendo que la clase Estudiante hereda de la clase Persona.

Cuando se implementan clases que utilizarán la herencia, se emplea la palabra protected en lugar de private para definir atributos o métodos donde es usada; será privada para cualquier clase externa, pero podrán ser utilizados como si fuera pública para cualquier clase derivada o clase hija.

La herencia se identifica al encontrar la relación ***es-un*** entre la nueva clase y la ya existente.

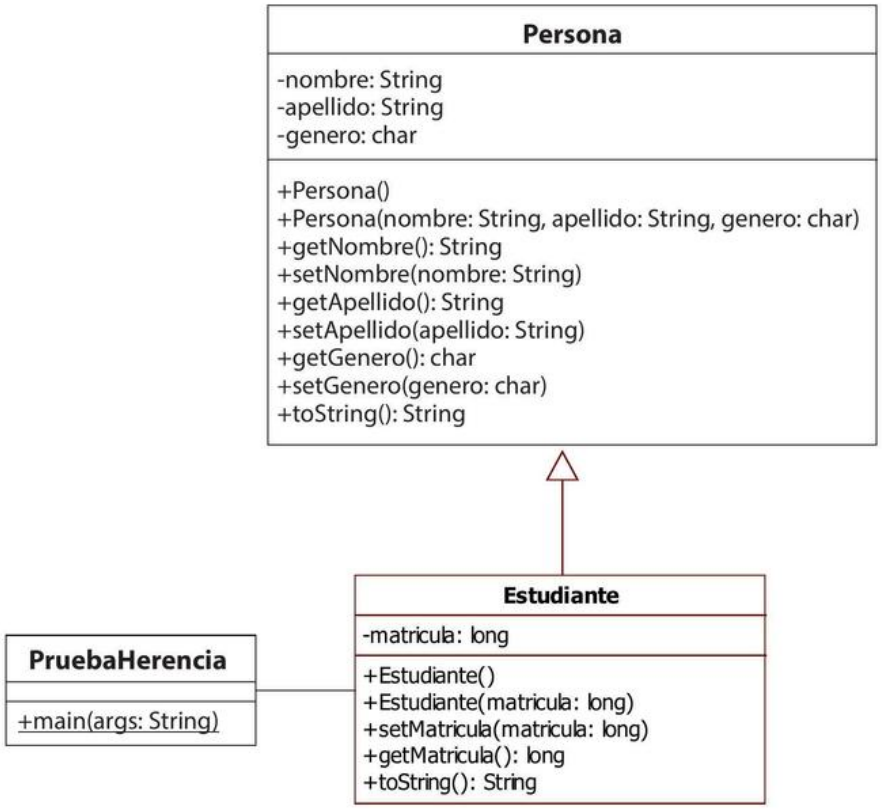
Ejemplos:

- Un estudiante ***es una*** Persona.
- Una acelga ***es una*** Verdura, que ***es un*** tipo de Vegetal.
- Un DVD ***es una*** unidad de almacenamiento, que a su vez ***es un*** tipo de componente de un computador.

Al decir que un elemento es un tipo de otro, se establece una especialización:

- Donde el elemento tiene las características generales compartidas y otras propias.
- La herencia es una manera de representar las características que se reciben del nivel más general.

Codificación de las clases Persona y Estudiante del ejemplo:



En las clases Persona y Estudiante se agruparon los elementos que pueden considerarse pertenecientes a esas entidades, al mismo nivel de abstracción, propiedad de la programación orientada a objetos denominada encapsulamiento.

```
public class Persona {
        protected String nombre;
        protected String apellido;
        protected char genero;

        public Persona() {
                this.nombre = "";
                this.apellido = "";
                this.genero = ' ';
        }
```

```
        public Persona(String nombre, String apellido, char genero) {
                this.nombre = nombre;
                this.apellido = apellido;
                this.genero = genero;
        }

        public String getNombre() {
                return nombre;
        }

        public void setNombre(String nombre) {
                this.nombre = nombre;
        }

        public String getApellido() {
                return apellido;
        }

        public void setApellido(String apellido) {
                this.apellido = apellido;
        }

        public char getGenero() {
                return genero;
        }
        public void setGenero(char genero) {
                this.genero = genero;
        }


        @Override
        public String toString() {
                return nombre+""+apellido;
        }
}


public class Estudiante extends Persona{
        private long matricula;
```

```
        public Estudiante() {
                this.matricula=0;
        }

        public Estudiante(long matricula) {
                this.matricula = matricula;
        }

        public void setMatricula(long matricula) {
                this.matricula = matricula;
        }

        public long getMatricula() {
                return matricula;
        }

        @Override
        public String toString() {
                return ""+matricula;
        }
}


public class PruebaHerencia {
        public static void main(String[] args) {
                Estudiante unEstudiante = new Estudiante();
                unEstudiante.setNombre("Alan");
                unEstudiante.setApellido("Brito");
                unEstudiante.setGenero('M');
                unEstudiante.setMatricula(12345);
                System.out.println("\nEstudiante");
                System.out.println("Matricula: " +
unEstudiante.getMatricula());
System.out.println("Nombre: " + unEstudiante.getNombre());
                System.out.println("Apellido: " +
unEstudiante.getApellido());
System.out.println("Genero: " + unEstudiante.getGenero());
        }
}
```

Al heredar de una clase base, se heredan tanto los atributos como los métodos, mientras que los constructores son utilizados, pero no heredados.

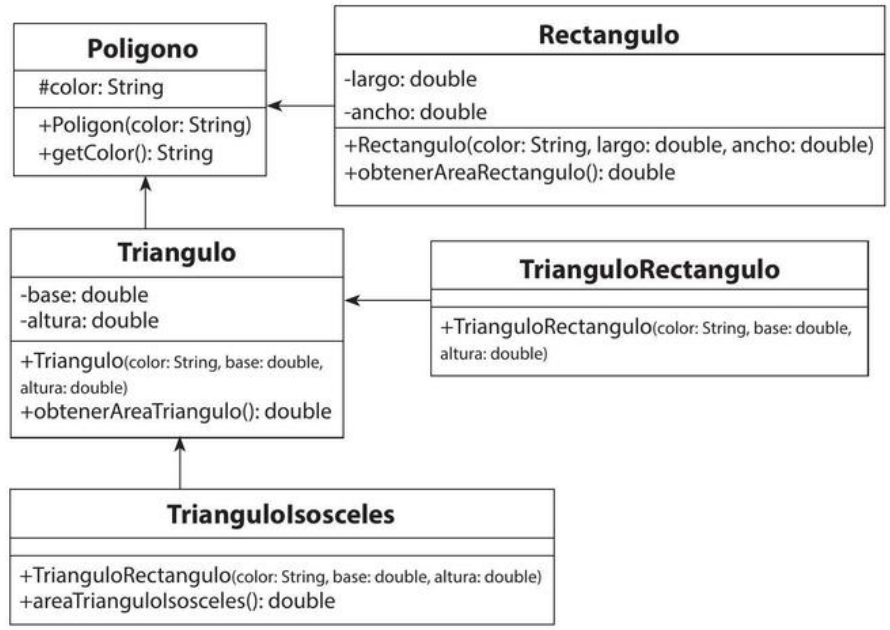
10.5 Sentencia super

- La palabra reservada super sirve para indicar que una variable o un método es de la superclase; super() llama al constructor de la clase de la que se hereda (extends). Es de tener en cuenta que todas las clases heredan en última instancia a la clase Object.
- El uso de super es opcional, pero, en caso de utilizarse, ha de ser obligatoriamente la primera sentencia del constructor. Si no se utiliza, el compilador inserta la sentencia super() al comienzo de cada método constructor de la subclase.
- super permite acceder, desde la subclase, a los métodos y las variables de la superclase.

Programa que calcula el área del rectángulo, de los triángulos rectángulo e isósceles, a partir de la base y la altura e imprimiendo el color de cada figura geométrica.

El modelamiento de las clases se realiza a partir de:

- Un rectángulo ***es un*** polígono.
- Un triángulo ***es un*** polígono.
- Un triángulo rectángulo ***es un*** triángulo.
- Un triángulo isósceles ***es un*** triángulo.
- Superclase Poligono con las subclases Triangulo y Rectangulo.
- Superclase Triangulo con las subclases TrianguloIsosceles y TrianguloRectangulo.




```
class Poligono {
        protected String color;
        public Poligono(String color ) {
                this.color=color;
        }
        public String getColor() {
                return color;
        }
}

class Rectangulo extends Poligono {
        private double largo;
        private double ancho;
        public Rectangulo(String color,double largo,double ancho) {
                super(color);
                this.largo=largo;
                this.ancho=ancho;
        }
        public double obtenerAreaRectangulo(){
                return largo*ancho;
```

```
        }
}


class Triangulo extends Poligono {
        private double base;
        private double altura;
        public Triangulo(String color, double base, double altura) {
                super(color);
                this.base=base;
                this.altura=altura;
        }

        public double obtenerAreaTriangulo(){
                return base*altura/2;
        }
}


public class TrianguloIsosceles extends Triangulo {
        public TrianguloIsosceles(String color, double base, double altura){
                super(color,base,altura);
        }
        public double areaTrianguloIsosceles(){
                double calculoArea= super.obtenerAreaTriangulo();
                return calculoArea;
        }
}


public class TrianguloRectangulo extends Triangulo {
        public TrianguloRectangulo(String color,double base,double altura){
                super(color,base,altura);
        }
}
public class Principal {
        /**
         * método principal, entrada de la aplicación en Java
         * @param args
         */
        public static void main(String[] args) {
```

```
                    double superficie;
                    //definición objeto rectangulo
                    Rectangulo rectangulo=new Rectangulo("Azul",5,4);
                    superficie=rectangulo.obtenerAreaRectangulo();
                    System.out.println("Este rectángulo es de color
"+rectangulo.getColor()+" de base = 5 y altura = 4 ");
                    System.out.println("El área es: "+superficie);
                    //definición objeto trianguloIsosceles
                    TrianguloIsosceles trianguloIsosceles=new
                    TrianguloIsosceles("Verde",6,8);
                    superficie=trianguloIsosceles.areaTrianguloIsosceles();
                    System.out.println("Este triangulo isósceles es de color
"+trianguloIsosceles.getColor()+" de base = 6 y altura = 8 ");
                    System.out.println("El área es: "+superficie);
                    //definición objeto trianguloRectangulo
                    TrianguloRectangulo trianguloRectangulo=new
                    TrianguloRectangulo("Rojo ",7,5);
                    superficie=trianguloRectangulo.obtenerAreaTriangulo();
                    System.out.println("Este triangulo Rectangulo es de color
"+trianguloRectangulo.getColor()+" de base = 7 y altura = 5 ");
                    System.out.println("El área es: "+superficie);
          }
}
```

10.6 Lecturas recomendadas

- Herencia por extensión.
- Herencia por restricción.
- Herencia múltiple.

10.7 Preguntas de revisión de conceptos

- Objetivos de la herencia.
- Cómo identificar que se debe aplicar herencia para resolver un problema.
- Tipos de herencia.
- Tipos de herencia en Java.
- A qué corresponde la clase Object en Java.

10.8 Ejercicios

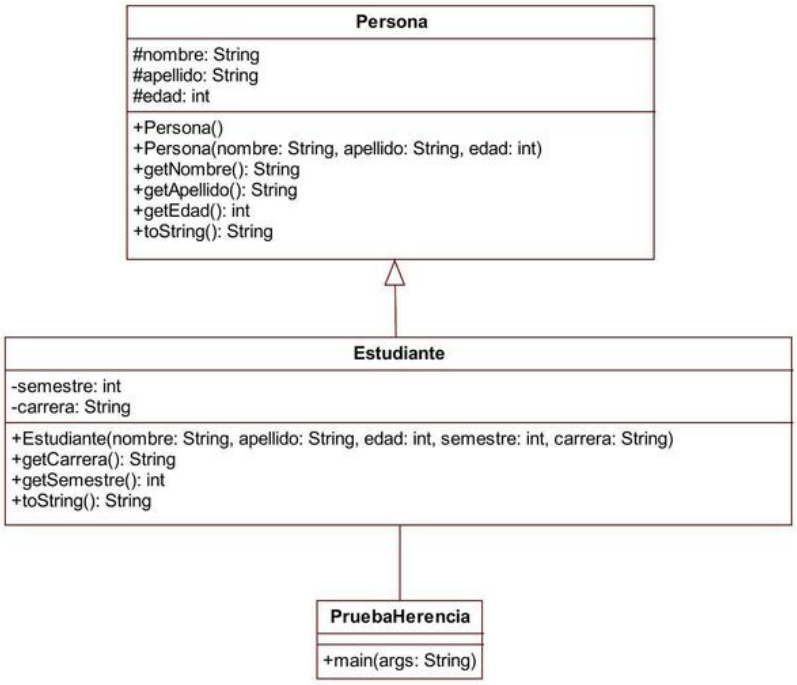
1. Crear el diagrama de clase para los siguientes requerimientos:

   - Implementar la clase base denominada Animal, con sus atributos y métodos particulares, por ejemplo, nombre.
   - Crear clases heredadas de Animal, como, por ejemplo: Mamíferos, Reptiles, entre otras.
   - Crear subclases de las creadas anteriormente, como Perro, León, Delfín.

2. Crear el diagrama de clase para el siguiente requerimiento:

   Las impresoras láser, de burbuja, 3D y de matriz de punto son subclases de la superclase Impresora. Los atributos generales de una impresora son la marca, la referencia, la velocidad, el precio y la resolución, mientras que sus operaciones son imprimir y forma de alimentar el papel. La clase Impresora es una generalización de las clases: ImpresoraLaser, ImpresoraBurbuja, Impresora3D e ImpresoraMatrizPunto.

3. Codificar en Java a partir del siguiente diagrama de clases:



4. Para cada uno de los siguientes enunciados, construir el diagrama de clases en UML y codificar en Java el diagrama de clases diseñado:

   a. En la universidad El Templo del Saber existen estudiantes, docentes, directivos, personal de mantenimiento; todos estos tienen diferentes propiedades comunes, como la identificación, nombre, apellidos, género, edad. Pero existen algunas características propias para el caso de los estudiantes, el semestre, entre otras. Y para los docentes, directivos y demás personal, el sueldo. Estas son las propiedades que se tendrán en cuenta para modelar el sistema.

   b. El banco El Ahorrador Feliz requiere la construcción de un sistema que le permita controlar las cuentas de sus clientes y para ello las clasifica en dos: cuentas de ahorros y cuentas corriente. Todas las cuentas del banco tienen los siguientes datos:

      - Número de cuenta (dato numérico).
      - Nombre del cliente (cadena).
      - Saldo (dato numérico con fracción decimal).

      Con cada cuenta se pueden realizar las siguientes operaciones:

      - Consultar datos: a través de sus propiedades.
      - Depositar: incrementa el saldo con la cantidad de dinero que se deposita.
      - Retirar: antes de hacer el retiro, se debe verificar la suficiencia de saldo y, en caso de aprobarlo, se disminuye el saldo.

      Las cuentas de ahorros presentan las siguientes características:

      - Porcentaje de interés mensual.
      - Método para depositar los intereses el primer día de cada mes.
      - Solamente se puede retirar un monto menor o igual al saldo.

      Las cuentas corrientes presentan las siguientes características:

      - Cobro del 4x1.000 en cada transacción
      - Se puede sobregirar hasta en $300.000 cuando el saldo no es suficiente para el retiro.

10.9 Referencias bibliográficas

Aguilar, L. J. (2004). *Fundamentos de programación algoritmos y estructura de datos*. Tercera edición. Ciudad: México. McGraw-Hill.

Barnes, D. y Kölling, M. (2007). *Programación orientada a objetos con Java*. Ciudad: Madrid. Prentice Hall.

Bertrand, M. (1999). *Construcción de software orientado a objetos*. Segunda edición. Ciudad: Madrid. Prentice Hall.

Booch, G. (1996). *Análisis y diseño orientado a objetos con aplicaciones*. Ciudad: México. Addison Wesley.

Deitel y Deitel (2012). *Cómo programar en Java*. Ciudad: México. Editorial Pearson (Prentice Hall).

Fowler, M. y Scott, K. (1999). *UML gota a gota*. Ciudad: México. Editorial Pearson.

Graig, L. (2003). *UML y patrones: una introducción al análisis y diseño orientado a objetos y al proceso unificado*. Segunda edición. Ciudad: México. Prentice Hall.

Rumbaugh, J., Jacobson, I. y Booch, G. (2005). *The Unified Modeling Language User Guide*. Ciudad: San Francisco. Addison-Wesley, Reading, Mass. ISBN-13: 078-5342267976, ISBN-10: 0321267974.

Villalobos, J. y Casallas, R. (2006). *Fundamentos de programación, aprendizaje activo basado en casos*. Ciudad: México. Editorial Pearson.

Capítulo 11

Polimorfismo

11.1 Temática a desarrollar

- Polimorfismo
- Polimorfismo de subtipado
- Sobrecarga de métodos
- Sobreescribir métodos

11.2 Introducción

El término polimorfismo hace referencia a que una característica de una clase puede tomar varias formas y en programación orientada a objetos es una propiedad que hace referencia a que más de un objeto puede crearse usando la misma clase base, por ejemplo, se tiene un teléfono base, con la capacidad de timbrar y tener un auricular, para luego obtener un teléfono digital, inalámbrico, con botones para marcar, y posteriormente un teléfono celular, entonces, polimorfismo es la capacidad del código de un programa para ser utilizado con diferentes tipos de datos u objetos.

El polimorfismo se puede establecer mediante la sobrecarga, que se refiere al uso del mismo identificador u operador en distintos contextos y con distintos significados. Java permite la sobrecarga de métodos, que es una de las formas con las cuales se implementa el polimorfismo; esto quiere decir que en una misma clase puedan existir métodos con el mismo nombre.

La herencia y el polimorfismo están estrechamente ligados. La herencia implica polimorfismo, el cual genera el polimorfismo de subtipado, que es la capacidad para redefinir un método en clases que se hereda de una clase base; a esto se le llama especialización. Por lo tanto, se puede llamar a un método

de objeto sin tener que conocer su tipo intrínseco: esto es polimorfismo de subtipado, por ejemplo, en un juego de ajedrez con los objetos rey, reina, alfil, caballo, torre y peón, cada uno heredando el objeto pieza. El método movimiento podría ser uno solo sin tener que verse conectado con cada tipo de pieza del ajedrez en particular.

La figura 26 detalla el concepto de polimorfismo.

11.3 Polimorfismo

Es la propiedad que indica que un elemento puede tomar diferentes formas. Para la programación orientada a objetos, el elemento puede ser un atributo, un método o un operador.

Por ejemplo, en Java, un caso muy común es el del operador más (+), el cual se puede utilizar para sumar números y para concatenar cadenas, ya que el operador está sobrecargado, de tal manera que, cuando se utiliza, Java reconoce el tipo de operación suma o concatenación respectivamente.

Para el caso de los métodos, existe una clase de polimorfismo que se denomina sobrecarga, específicamente sobrecarga de métodos, y que se define como el uso de varios métodos con el mismo nombre y cuyo contenido es diferente.

11.4 Sobrecarga de métodos

La sobrecarga consiste en implementar dos o más métodos que compartan el mismo nombre dentro de la misma clase, pero con parámetros diferentes. Los métodos se diferencian por su firma.

Java permite la sobrecarga de métodos e identifica entre métodos con diferentes firmas, evitando la necesidad de crear métodos diferentes que en esencia hacen lo mismo. También hace posible que los métodos se comporten de manera distinta basados en los argumentos que reciben.

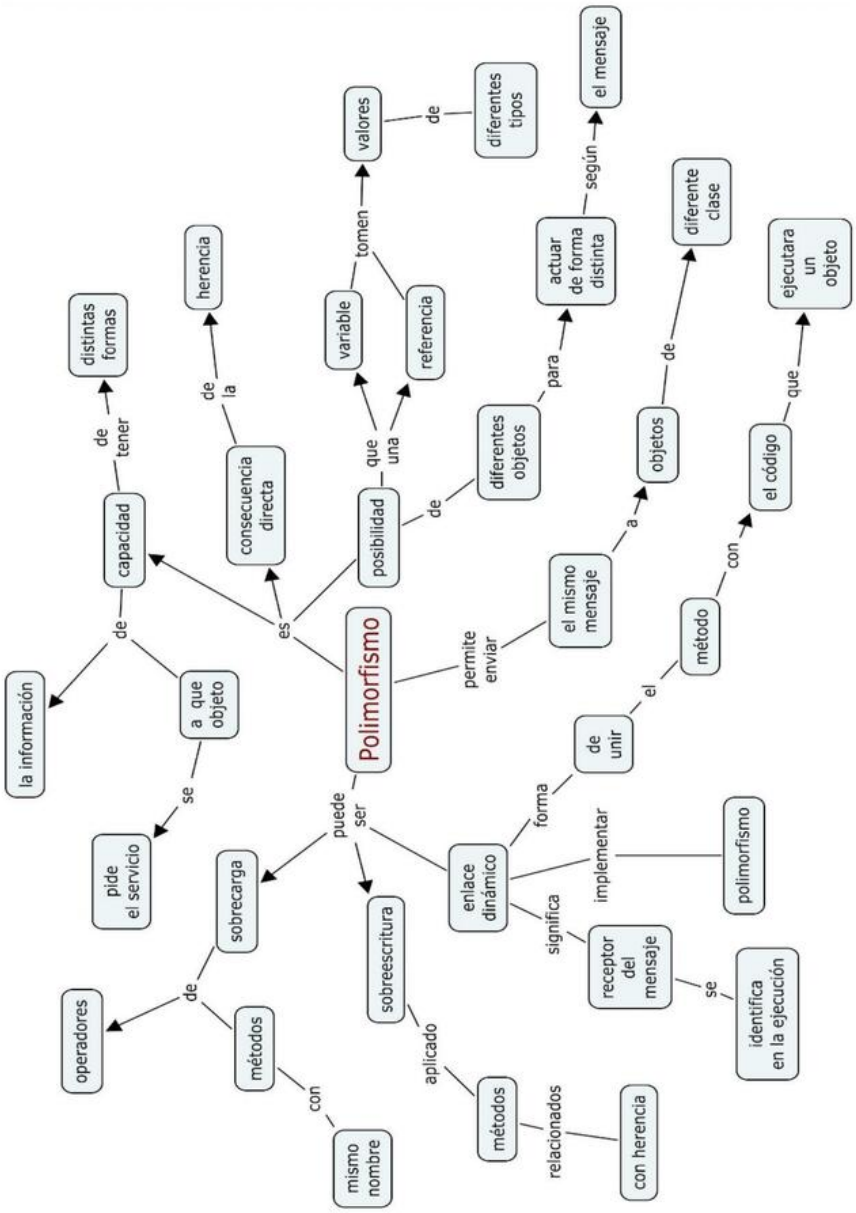


Figura 26.
Mapa mental de polimorfismo

Ejemplos:

1. Sobrecarga del método actualizarDato.

```
public class Actualizacion {
        void actualizarDato(int x) {

        }

        void actualizarDato (String x) {

        }

        void actualizarDato (long x) {

        }


        void actualizarDato (float x) {

        }

        void actualizarDato (double x) {

        }
}
```

Se puede observar que el tipo de parámetro es diferente en cada método implementado, como, por ejemplo: int, long, String, float, double.

2. Sobrecarga del método obtenerSuma, el cual cambia el número y tipo de parámetros:

```
public class Suma{
        /** Métodos sobrecargados */

        double obtenerSuma (String x, String y) {
                ...
        }
```

```
        double obtenerSuma(double x, double y) {
                ...
        }

        double obtenerSuma (double x, double y, double z) {
                ...
        }
}
```

En la clase Suma, se definen tres métodos con el mismo nombre obtenerSuma, pero con listas de argumentos diferente. La diferencia está en el número y el tipo de argumentos en cada método.

Programas de ejemplo:

Sobrecarga de métodos para calcular producto, el cual cambia el número y el tipo de parámetros.

```
/**
 * Clase que modela la sobrecarga de métodos por el número
 * de parámetros de estos o el tipo de dato que recibe.
 * @version 1.0
 */

public class SobreCargaProducto {
        public int obtenerProducto(int x, int y){
                return x*y;
        }
        public int obtenerProducto(int x, int y, int z){
                return x*y*z;
        }
        public int obtenerProducto(String str1,String str2){
                int x=convertirEntero(str1);
                int y=convertirEntero(str2);
                return x*y;
        }
        private int convertirEntero(String numero){
                return Integer.parseInt(numero);
        }
}
```

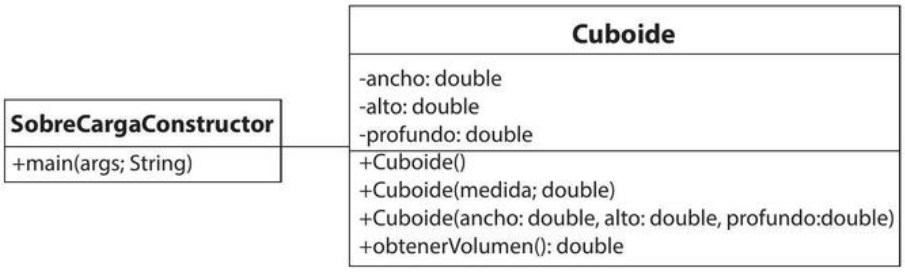
```
/**
 * clase que modela los datos para la sobrecarga de los métodos
 * @version 1.0
 */
public class MainSobrecarga {
        public static void main (String []args){
                int a = 2;
                int b = 4;
                int c = 5;
                String str1 = "8";
                String str2 = "7";
                SobreCargaProducto sp= new SobreCargaProducto();
                System.out.println("Método con dos parámetros: "+
                a+" x "+b+" = "+sp.obtenerProducto(a,b));
                System.out.println("Método con tres parámetros: "+
                a+" x "+b+" x "+c+" = "+sp.obtenerProducto(a,b,c));
                System.out.println("Método con parámetros de tipo String: "+
                str1+" x "+str2+" = "+sp.obtenerProducto(str1,str2));
        }
}
```

11.5 Sobrecarga de constructores

Una clase puede tener más de un constructor y los constructores se diferencian por la cantidad, el tipo y el orden de los parámetros.

Sobrecarga del constructor de la clase Cuboide, la cual está conformada por tres constructores, con un número de diferentes parámetros:

Diagrama de clases:



La implementación del diagrama de clases en lenguaje Java es la siguiente:

```
public class SobreCargaConstructor {
        public static void main(String[] args) {
                //se definen tres instancias de la clase Cubo
          Cuboide cubo1 = new Cuboide();
          Cuboide cubo2 = new Cuboide(7);
          Cuboide cubo3 = new Cuboide(10, 20, 15);
          //se define el atributo volumen de tipo double
          double volumen = 0.0;
          volumen = cubo1.volumen();
          System.out.println("El volumen de cubo1 es " + volumen);
          volumen = cubo2.volumen();
          System.out.println("El volumen de cubo2 es " + volumen);
          volumen = cubo3.volumen();
          System.out.println("El volumen de cubo3 es " + volumen);
        }
}

/**
 * clase que modela la sobrecaraga de constructores para calcular el volumen
del cuboide
*/
public class Cuboide {
 private double ancho;
 private double alto;
 private double profundo;

 /**
  * Constructor que inicializa valor de los atributos por defecto
```

```
 * las dimensiones del cubo no están definidas
 */
 public Cuboide() {
            this.ancho = 0;
            this.alto = 0;
            this.profundo = 0;
 }

 /**
 * Constructor con un parámetro
 * @param medida
 */
 public Cuboide(double medida) {
  this.ancho = medida;
  this.alto = medida;
  this.profundo = medida;
 }

 /**
 * Constructor con tres parámetros
 * @param ancho
 * @param alto
 * @param profundo
 */
 public Cuboide(double ancho, double alto, double profundo) {
            this.ancho = ancho;
            this.alto = alto;
            this.profundo = profundo;
           }

 /**
 * método que retorna el volumen del cuboide
 * @return volumen
 */
 public double obtenerVolumen() {
  return ancho * alto * profundo;
 }
}
```

Al sobrecargar los constructores de la clase Cuboide, se crean tres objetos (cubo1, cubo2, cubo3) con diferentes inicializaciones:

- Cuboide cubo1 = new Cuboide();
- Cuboide cubo2 = new Cuboide(7);
- Cuboide cubo3 = new Cuboide(10, 20, 15);

Estos, al ejecutar el programa, imprimen:

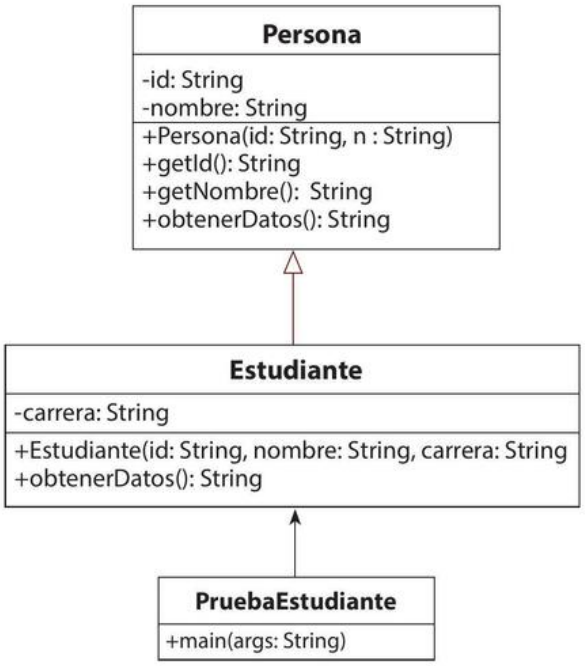
El volumen de cubo1 es: 0.0.
El volumen de cubo2 es: 343.0.
El volumen de cubo es: 3000.0.

11.6 Sobreescritura de métodos

Cuando una subclase implementa un método que fue definido en la superclase con el mismo nombre, la misma cantidad y los mismos tipos de parámetros, en este caso, se presenta la sobreescritura de métodos, que consiste en la implementación de métodos previamente definidos en la superclase, con igual firma.

Al heredar los métodos de una clase, se tiene la opción de sobreescribirlos para darles una funcionalidad distinta a la que tenían. Es decir, se toma el método de la clase padre en la clase hija y se realizan ajustes al código que se encuentra dentro de las llaves que definen el cuerpo del método, según sean los requerimientos, y a esto se le llama sobreescritura.

1. Ejercicio conformado por las clases Persona y Estudiante, que implementa el método obtenerDatos, método que es heredado de la clase Persona y sobreescrito en la clase Estudiante.




```
public class Persona {
        private String id;
        private String nombre;
        public Persona(String id, String nombre){
                this.id = id;
                this.nombre = nombre;
        }
        public String getId(){
                return id;
        }
        public String getNombre(){
                return nombre;
        }
        public String obtenerDatos(){
                return id +""+ nombre;
        }
}

public class Estudiante extends Persona {
```

```
        private String carrera;
        public Estudiante(String id, String        nombre, String carrera){
                super(id,nombre);
                this.carrera = carrera;
        }
        public String obtenerDatos(){
                return getId() + getNombre() + carrera;
        }
}

public class PruebaEstudiante {
        public static void main(String[ ] args) {
           Persona unEstudiante = new Estudiante("10020","Carlos
         Rodriguez","Ingenieria de Sistemas");
           System.out.println( unEstudiante.obtenerDatos());
        }
}
```

2. Cada clase de Java hereda de la clase Object y la clase Object tiene varios métodos, dentro de los que está el método toString().

```
public class Persona{
  private String nombre;
  public Persona(String n){
       this.nombre = n;
   }
  //aquí se sobreescribe el método
  @Override
  public String toString(){
       return nombre;
  }
}
```

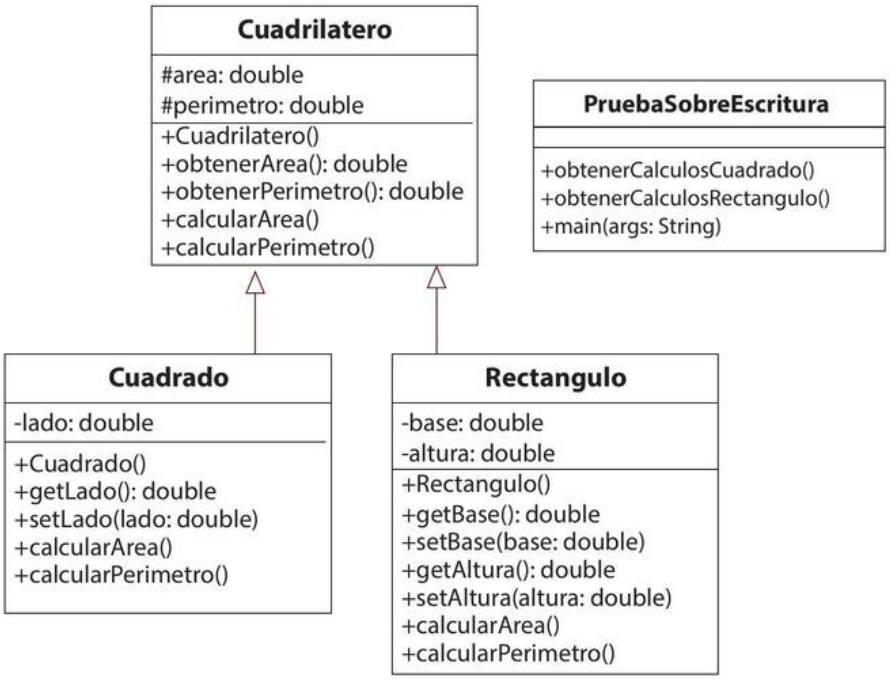
La clase Persona sobreescribe el método toString indicando que devuelve el nombre de la persona:

```
public class Prueba {
        public static void main(String[] args) {
                Persona p = new Persona("Mercedes");
                System.out.println("Nombre a partir del método
"+p.toString());
```

```
        System.out.println("Nombre a partir del objeto "+p);
        }
}
```

Se crea un objeto p de la clase Persona; al imprimir el objeto, por medio de la sentencia System.out.println("Nombre a partir del objeto "+p);, se envía, como parámetro, un objeto que llama al método toString e imprime "Mercedes". El proceso es el mismo cuando se invoca con el objeto p.toString().

3. A partir de la relación de herencia de la clase base Cuadrilatero, que define los métodos calcularArea y calcularPerimetro, las subclases Cuadrado y Rectangulo redefinen estos métodos, ya que el cálculo del área y el perímetro de cada una de estas dos figuras geométricas es diferente.




```
public class Cuadrilatero {
        protected double area;
        protected double perimetro;
```

```
        public Cuadrilatero() {
                this.area=0;
                this.perimetro=0;
        }
        public double obtenerArea() {
                return area;
        }
        public double obtenerPerimetro() {
                return perimetro;
        }
        public void calcularArea(){}
        public void calcularPerimetro(){}

}

public class Cuadrado extends Cuadrilatero{
        private double lado;

        public Cuadrado() {
                this.lado = 0;
        }
        public double getLado() {
                return lado;
        }
        public void setLado(double lado) {
                this.lado = lado;
        }

        public void calcularArea() {
                this.area = lado * lado;
        }
        public void calcularPerimetro() {
                this.perimetro = 4 * lado;
        }
}

public class Rectangulo extends Cuadrilatero{
        private double base;
        private double altura;
```

```
        public Rectangulo() {
                this.base = 0;
                this.altura = 0;
        }
        public double getBase() {
                return base;
        }

        public void setBase(double base) {
                this.base = base;
        }
        public double getAltura() {
                return altura;
        }
        public void setAltura(double altura) {
                this.altura = altura;
        }
        public void calcularArea() {
                this.area = base * altura;
        }

        public void calcularPerimetro() {
                this.perimetro = (2 * base)+(2 * altura);
        }
}

public class PruebaSobreEscritura {
                public void obtenerCalculosCuadrado(){
                System.out.println("\nEl Cuadrado\n");
                Cuadrado unCuadrado = new Cuadrado();
                unCuadrado.setLado(3);
                unCuadrado.calcularPerimetro();
                unCuadrado.calcularArea();
                unCuadrado.calcularPerimetro();
                unCuadrado.calcularArea();
                System.out.println("Lado "+unCuadrado.getLado());
                System.out.println("Perímetro del cuadrado:
"+unCuadrado.obtenerPerimetro());
                System.out.println("Área del cuadrado:
"+unCuadrado.obtenerArea());
        }
```

```
        public void obtenerCalculosRectangulo(){
                System.out.println("\nEl Rectangulo\n");
                Rectangulo unRectangulo = new Rectangulo();
                unRectangulo.setBase(3);
                unRectangulo.setAltura(4);
                unRectangulo.calcularPerimetro();
                unRectangulo.calcularArea();
                System.out.println("Base: "+unRectangulo.getBase());
                System.out.println("Altura: "+unRectangulo.getAltura());
                System.out.println("Perímetro del rectángulo:
"+unRectangulo.obtenerPerimetro());
                System.out.println("Área del rectángulo:
"+unRectangulo.obtenerArea());
        }

        public static void main(String[] args) {
        //se define el objeto pse de la clase PruebaSobreEscritura
                PruebaSobreEscritura pse = new PruebaSobreEscritura();
                //el objeto pse invoca los métodos
                pse.obtenerCalculosCuadrado();
                pse.obtenerCalculosRectangulo();
        }
}
```

11.7 Enlace dinámico

Un mensaje con una determinada firma puede enviarse a cualquier objeto que lo incluya en su interfaz, pero como diferentes objetos pueden implementar un mismo método de manera distinta, el resultado del mensaje no depende en sí del mismo mensaje, sino de su receptor.

El enlace dinámico significa que el receptor del mensaje se determina durante la ejecución del programa.

Polimorfismo y enlace dinámico no son exactamente lo mismo: el enlace dinámico es el mecanismo que hace posible implementar el polimorfismo.

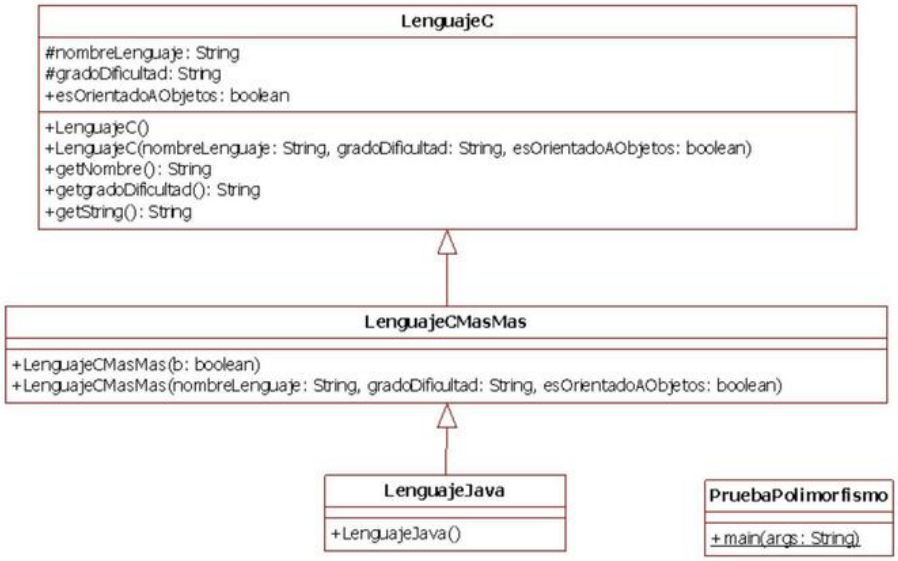
Los métodos polimórficos se utilizan cuando adoptan diferentes formas de implementación según el tipo de objetos, cumpliendo el mismo objetivo. Por

ejemplo, lanzar puede ser implementado para un objeto balón de: fútbol, baloncesto, voleibol o mover una ficha de un tablero de ajedrez.

Ejemplos:

El lenguaje C es uno de los lenguajes de programación más populares que existen hoy en día. El lenguaje C++ es un lenguaje derivado del C, pero mucho más complejo, ya que incluye la programación orientada a objetos y el lenguaje de programación Java es un lenguaje orientado a objetos, con una sintaxis similar a C++, pero con algunos conceptos de diseño diferentes.

- Lenguaje C: es un lenguaje de Programación que tiene características generales como un nombre, grado de dificultad y no es orientado a objetos.
- C++: es un lenguaje de programación orientada a objetos.
- C++: es una derivación del lenguaje C.
- Java: es un lenguaje de programación orientada a objetos.



```
public class LenguajeC  {
        protected String nombreLenguaje;
        protected String gradoDificultad;
        public boolean esOrientadoAObjetos;

        //sobrecarga de constructores
        /**
         * constructor vacio
         */
        public LenguajeC(){
                this.nombreLenguaje="C";
                this.gradoDificultad = "BAJO";
                this.esOrientadoAObjetos=false;
        }

public LenguajeC(String nombreLenguaje, String gradoDificultad,
boolean esOrientadoAObjetos){
                this.nombreLenguaje=nombreLenguaje;
                this.gradoDificultad = gradoDificultad;
                this.esOrientadoAObjetos = esOrientadoAObjetos;
        }
        public String getNombre(){
                return nombreLenguaje;
        }
        public String getgradoDificultad(){
                return gradoDificultad;
        }

        public String getString(){
                String s = nombreLenguaje;
                if(¡esOrientadoAObjetos){
                        s+= " No";
                }
                s += " es un Lenguaje de Programación Orientado a Objetos ";
                s+="\ny tiene un grado de dificultad de "+ gradoDificultad;
                return s;
        }
}

public class LenguajeCMasMas extends LenguajeC{
        //sobrecarga de constructores
```

```
                    public LenguajeCMasMas(boolean b){
                    //se inicializan atributos de la superclase LenguajeC
                    super("C++","ALTO",true);
}

public LenguajeCMasMas(String nombreLenguaje, String
gradoDificultad, boolean esOrientadoAObjetos){
                    super(nombreLenguaje, gradoDificultad,
          esOrientadoAObjetos);
          }
}
public class LenguajeJava extends LenguajeCMasMas {
          public LenguajeJava(){
                    //true, porque se indica que es orientado a objetos
                    super("Java","MEDIO",true);
          }
}
public class PruebaPolimorfismo {
          public static void main(String args[]){
                    LenguajeC unLenguaje;
                    unLenguaje = new LenguajeC();
                    System.out.println("\n"+unLenguaje.getString());
                    unLenguaje = new LenguajeCMasMas(true);
                    System.out.println("\n"+unLenguaje.getString());
                    unLenguaje = new LenguajeJava();
                    System.out.println("\n"+unLenguaje.getString());
          }
}
```

Aunque los objetos se accedan con referencias de superclase, el método que se ejecuta es el de la clase real del objeto.

11.8 Lecturas recomendadas

- Polimorfismo.
- Sobrecarga.
- Enlace dinámico.

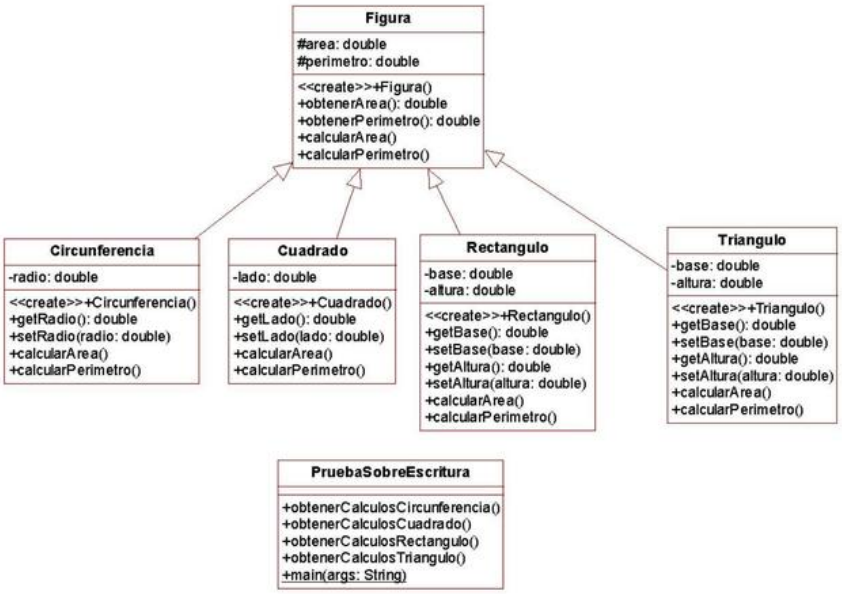
11.9 Preguntas de revisión de conceptos

- Objetivos del polimorfismo.
- Cómo identificar que se debe aplicar polimorfismo para resolver un problema.

11.10 Ejercicios

1. Construir la aplicación en Java a partir de la relación de herencia de la superclase Figura que define los métodos calcularArea y calcularPerimetro. Las subclases Circunferencia, Cuadrado, Triangulo y Rectangulo redefinen estos métodos, ya que el cálculo del área y el perímetro de cada una de estas figuras geométricas es diferente.

Diagrama de clases:



2. Construir una aplicación en Java que aplique los conceptos de sobrecarga de constructores para inicializar las dimensiones de un rectángulo, teniendo en cuenta las siguientes indicaciones:

a. El primer constructor de Rectangulo inicializa un nuevo Rectangulo con valores por defecto (por ejemplo altura = 0.0, ancho = 0.0).
b. El segundo constructor inicializa el nuevo Rectangulo con la altura y ancho a partir de datos ingresados por teclado.

3. Se tiene la información del estudiante conformada por el código, nombre, apellido y edad. Se requiere construir una aplicación en la cual se implemente la sobrecarga de los métodos constructores.

4. Consultar el método toString() de Java y aplicarlo para el siguiente ejercicio:

La empresa El Viajero ha solicitado la creación de un sistema donde quiere tener representados a todos sus productos en clases de Java, también solicitó que fuera realizado de tal manera (aplicando la abstracción) que, si es necesario agregar un nuevo producto, no fuera un impacto muy grande para el sistema. La empresa se especializa en todo tipo de transportes y maquinarias; actualmente tiene en sus vitrinas los siguientes productos:

- Buses
- Camiones
- Helicópteros
- Aviones

11.11 Bibliografía

Bertrand, M. (1999). *Construcción de software orientado a objetos*. Segunda edición. Ciudad: Madrid. Prentice Hall.

Booch, G. (1996). *Análisis y diseño orientado a objetos con aplicaciones*. Ciudad: México. Addison Wesley.

Deitel y Deitel (2012). *Cómo programar en Java*. Ciudad: México. Editorial Pearson (Prentice Hall).

Fowler, M. y Scott, K. (1999). *UML gota a gota*. Ciudad: México. Editorial Pearson.

Graig, L. (2003). *UML y patrones: una introducción al análisis y diseño orientado a objetos y al proceso unificado*. Segunda edición. Ciudad: México. Prentice Hall.

Rumbaugh, J., Jacobson, I. y Booch, G. (2005). *The Unified Modeling Language User Guide*. Ciudad: San Francisco. Addison-Wesley, Reading, Mass. ISBN-13: 078-5342267976, ISBN-10: 0321267974.

Villalobos, J. y Casallas, R. (2006). *Fundamentos de programación, aprendizaje activo basado en casos*. Ciudad: México. Editorial Pearson.

Capítulo 12

Interfaz y clases abstractas

12.1 Temática a desarrollar

- Interfaz
- Clases abstractas

12.2 Introducción

En este capítulo se abordan los temas de interfaz y clase abstracta, donde una interfaz es una plantilla para la construcción de clases la cual está conformada por la declaración de métodos sin implementar que especifican un protocolo de comportamiento para una o varias clases. Lo anterior permite organizar la programación y a su vez obliga a que ciertas clases utilicen los mismos métodos (nombres y parámetros) y se establecen relaciones entre clases que no estén relacionadas.

En Java, una clase puede implementar una o varias interfaces, en ese caso, la clase debe proporcionar la declaración y definición de todos los métodos de cada una de las interfaces o bien declararse como clase abstract. Por otro lado, una interfaz puede emplearse también para declarar constantes que luego puedan ser utilizadas por otras clases.

Las clases abstractas son un tipo de clases que permiten crear métodos vacíos, es decir, no tienen cuerpo de declaración, no tienen las llaves { } ni código dentro de ellos y deben estar precedidos por la palabra clave abstract. Si una clase contiene uno o más métodos abstractos, esta clase debe ser abstracta.

12.3 Interfaz

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

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

Diagrama UML

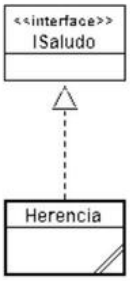
```
public interface TransporteAereo {
        public void despegar()  ;
        public void aterrizar() ;
}
```

Cualquier clase que declare que implementa una determinada interfaz debe comprometerse a implementar todos y cada uno de los métodos que esa interfaz define. Esa clase, además, pasará a pertenecer a un nuevo tipo de datos extra que es el tipo de la interfaz que implementa.

```
public class Viaje implements TransporteAereo {
        public void despegar( ) {
                System.out.println("despegar");
    }
        public void aterrizar(){
                System.out.println("aterrizar");
    }
}
```

En este ejemplo, se define la interface ISaludo, que tiene un atributo y dos métodos, y la clase Herencia, que implementa los métodos de la interfaz.

```
public interface ISaludo {
        public static final String nombre = "Hola soy una Interface";
        public abstract void saludar();
        public abstract String getNombre();

}
public class Herencia implements ISaludo{
        public Herencia (){
            saludar();
        }
        @Override
        public String getNombre(){
                return nombre;
        }
        @Override
        public void saludar(){
                System.out.println("Hola "+getNombre());
        }

        public static void main(String... args){
                Herencia hereda = new Herencia();
        }
}
```



Las interfaces actúan, por tanto, como tipos de clases. Se pueden declarar variables que pertenezcan al tipo de la interfaz y se pueden declarar métodos cuyos argumentos sean del tipo de la interfaz.

Una clase puede implementar tantas interfaces como desee, pudiendo, por tanto, pertenecer a tantos tipos de datos diferentes como interfaces implemente. En este sentido, las interfaces suplen la carencia de herencia múltiple de Java (y además corrigen o evitan todos los inconvenientes que del uso de esta se derivan).

Una interfaz es una lista de métodos donde solo están las cabeceras de métodos, sin implementación, que define un comportamiento específico para un conjunto de objetos. Algunas de sus características son:

- La interfaz es public o default, tal como se observa a continuación:

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

- Los atributos de la interfaz son public static final, así como se presenta en el siguiente ejemplo:

```
public interface PuntosCardinales {
        public static final int NORTE = 1;
        public static final int SUR = 2;
        public static final int ORIENTE  = 3;
        public static final int OCCIDENTE  = 4;

}
```

Una interfaz puede contener identificadores de variables constantes; estas variables constantes se heredan por cualquier clase que implemente la interfaz. El siguiente código muestra un ejemplo:

```
public interface IdentificadorInterfaz {
        public static final int VALOR_MAXIMO = 10;
}
```

- Todos los métodos de una interfaz son métodos public abstract y las firmas de métodos son sin implementación, como se observa en las siguientes líneas de código:

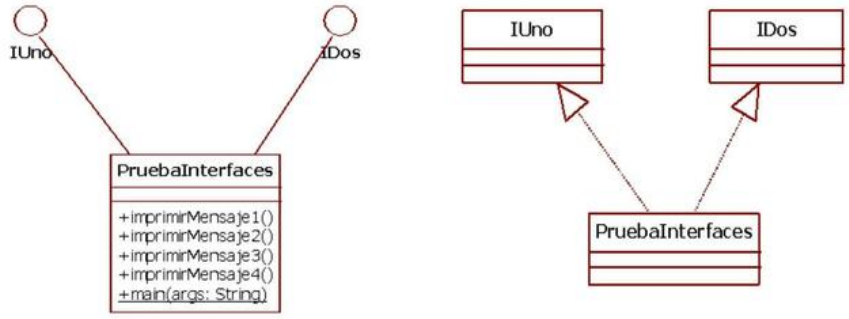
```
public interface Figura {

        public abstract double calcularArea( );
        public abstract double calcularPerimetro( );

}
```

- Permite simular herencia múltiple.

```
public interface IUno {
        public abstract void imprimirMensaje1();
        public abstract void imprimirMensaje2();
}
```

```
public interface IDos {
        public abstract void imprimirMensaje3();
        public abstract void imprimirMensaje4();
}

public class PruebaInterfaces implements IUno, IDos {

        @Override
        public void imprimirMensaje1() {
                 System.out.println("Mensaje 1");
        }

        @Override
        public void imprimirMensaje2() {
                System.out.println("Mensaje 2");
        }

        @Override
        public void imprimirMensaje3() {
                System.out.println("Mensaje 3");
        }

        @Override
        public void imprimirMensaje4() {
                System.out.println("Mensaje 4");
        }

        public static void main(String[] args) {
                PruebaInterfaces prueba = new PruebaInterfaces();
                prueba.imprimirMensaje1();
                prueba.imprimirMensaje2();
                prueba.imprimirMensaje3();
                prueba.imprimirMensaje4();
        }

}
```

Diagrama de clases:



El uso de las interfaces se puede aplicar en:

- La revelación de la interfaz de la programación de un objeto (funcionalidad del objeto) sin presentar su implementación (encapsulado).
- La implementación puede cambiar sin afectar el llamado de la interfaz, que no necesita la implementación en Java.
- La implementación de métodos de similares comportamientos en clases sin relacionar.
- El modelado de herencia múltiple, imponiendo conjuntos múltiples de comportamientos a la clase.

12.4 Clases abstractas

Las clases abstractas se declaran con la palabra reservada abstract. Pueden definir métodos y variables, pero no se pueden usar directamente para crear objetos usando new. Algunas de las características son:

- A partir de una clase abstracta, se tiene la posibilidad de declarar clases, las cuales definen los métodos sin tener que implementarlos.
- Las subclases de una clase abstracta deben sobreescibir los métodos abstractos definidos.
- Una clase abstracta puede incluir métodos implementados y métodos abstractos.

Es necesario crear una clase que extienda la clase abstracta e implemente todos los métodos que en la clase abstracta hayan sido declarados abstract. Por ejemplo:

```
public abstract class ClaseAbstracta {
        public abstract void métodoUno();
        public String toString() {
                return "Clase Abstracta";
        }
}
```

El métodoUno() es un método abstracto que incluye la implementación (no tiene código); solo se define la firma del método (nombre y parámetros).

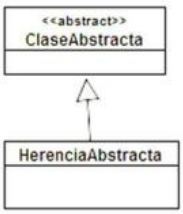
El método toString(), en cambio, no es un método abstracto y sí incluye implementación.

```
class SubClase extends ClaseAbstracta {
        public void métodoUno() {
                System.out.println("Método uno");
        }
}
```

La clase derivada SubClase implementa la abstracción, pues deriva de ClaseAbstracta e implementa métodoUno().

En la ClaseAbstracta, se define el método abstracto despedir, que es implementado en la clase HerenciaAbstracta.

```
public abstract class ClaseAbstracta{
        public String nombre;

        public void saludar(){
                System.out.println("Hola "+nombre);
        }
```




```
        public abstract void despedir();
}
public class HerenciaAbstracta extends ClaseAbstracta{

        public HerenciaAbstracta(){
                obtenerNombre("Estudiante");
                saludar();
                despedir();
        }


        public void obtenerNombre(String nombre){
                super.nombre = nombre;
        }

        @Override
        public void despedir() {
                System.out.println("Hasta pronto "+super.nombre);

        }


        public static void main(String... args){
                HerenciaAbstracta ha = new HerenciaAbstracta();
        }
}
```

12.5 Preguntas de revisión de conceptos

- ¿Cuáles son las similitudes y las diferencias entre una clase abstracta y una interfaz?
- ¿Cuándo se recomienda utilizar una interfaz y una clase abstracta?
- Cuando una clase implementa varias interfaces, ¿se puede considerar que es herencia múltiple?

12.6 Ejercicios

1. Construir una aplicación con el respectivo diagrama de clases en UML y la codificación Java de una aplicación que permita dar respuesta a los siguientes requerimientos:

   Que permita calcular el área y el perímetro para las siguientes figuras geométricas, para lo cual se recomienda implementar la interfaz IFigura.

```
public interface IFigura {
        public double calcularArea( );
        public double calcularPerimetro( );
}
```

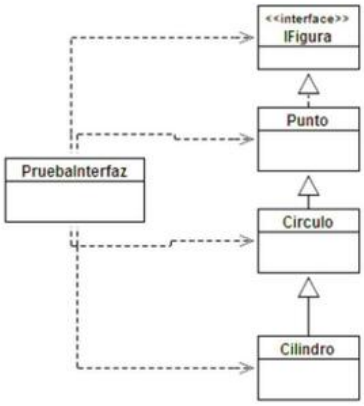
Las clases que implementa esta interfaz son:

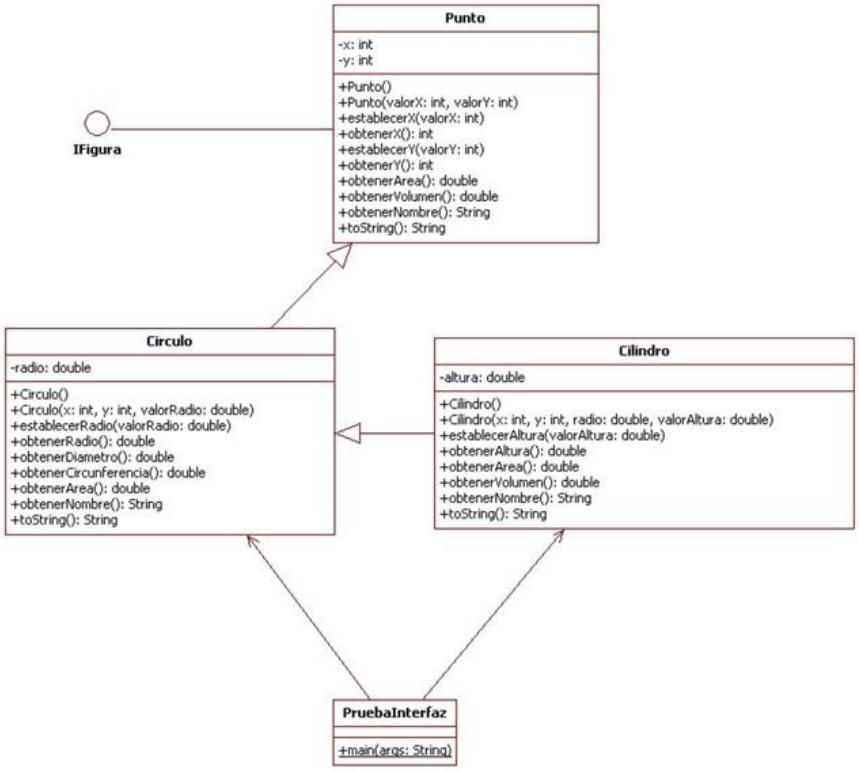
- La circunferencia
- El rectángulo
- El triángulo equilátero

2. Construir una aplicación a partir del siguiente diagrama de clases en UML realizando la codificación Java de una aplicación que permita dar respuesta a los siguientes requerimientos:

   Que permita calcular el área, el volumen y obtener el nombre de la figura, para lo cual se recomienda implementar la interfaz IFigura.

```
public interface IFigura
{
  public double obtenerArea();
  public doubleobtenerVolumen();
  public String obtenerNombre();
}
```



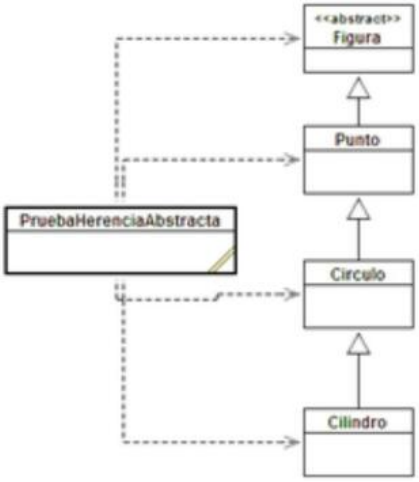


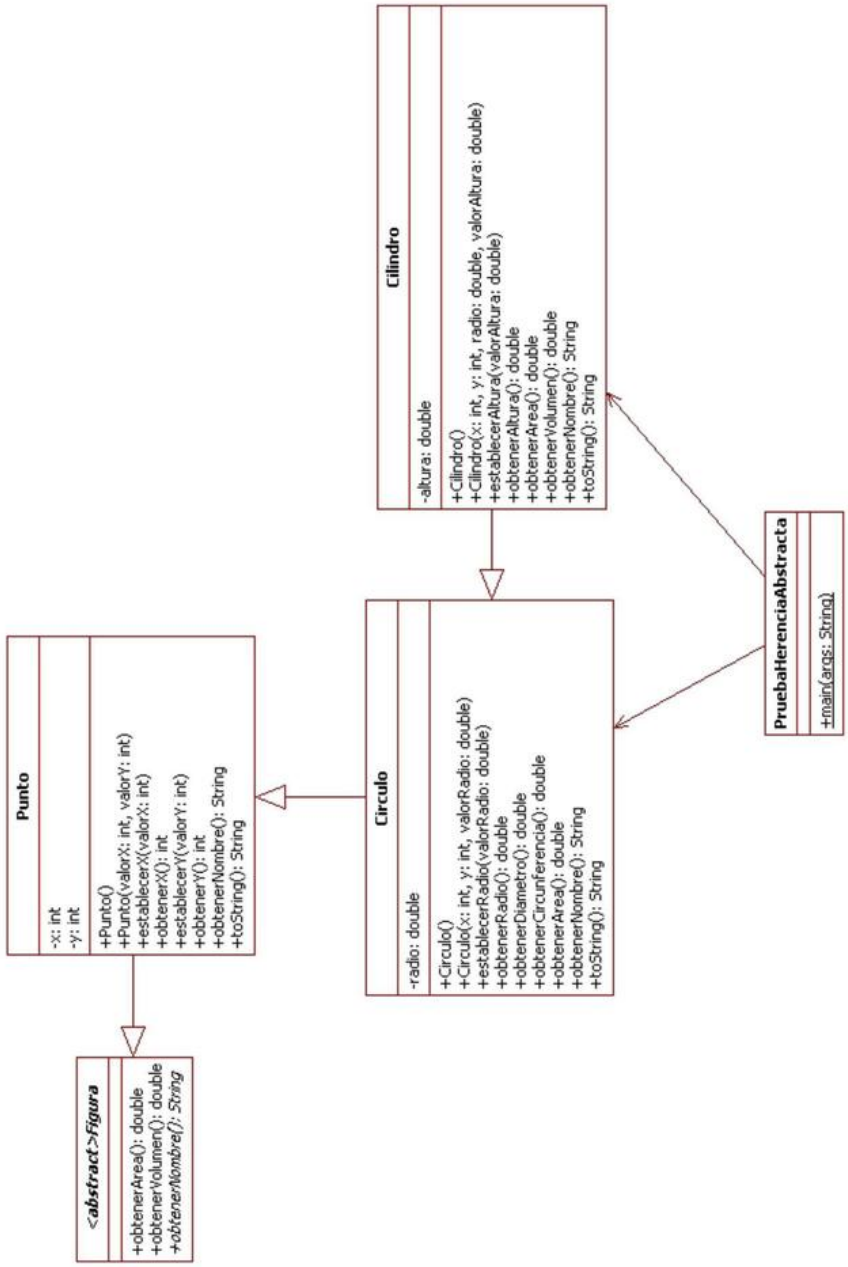
3. Construir una aplicación a partir del siguiente diagrama de clases en UML realizando la codificación Java de una aplicación que permita dar respuesta a los siguientes requerimientos:

   Que permita calcular el área, el volumen y obtener el nombre de la figura, para lo cual se recomienda implementar la clase abstracta Figura.

```
public abstract class Figura {

  // retorna el área de la figura; 0.0 de manera predeterminada
  public double obtenerArea()  {
    return 0.0;
  }
```

```
  // retorna el volumen de la figura; 0.0 de forma predeterminada
  public double obtenerVolumen()  {
    return 0.0;
  }

  // método abstracto, sobreescrito por las subclases
  public abstract String obtenerNombre();

}
}
```




Cilindro
-altura: double
+Cilindro()
+Cilindro(x: int, y: int, radio: double, valorAltura: double)
+establecerAltura(valorAltura: double)
+obtenerAltura(): double
+obtenerArea(): double
+obtenerVolumen(): double
+obtenerNombre(): String
+toString(): String
Punto
-x: int
-y: int
+Punto()
+Punto(valorX: int, valorY: int)
+establecerX(valorX: int)
+obtenerX(): int
+establecerY(valorY: int)
+obtenerY(): int
+obtenerNombre(): String
+toString(): String
Circulo
-radio: double
+Circulo()
+Circulo(x: int, y: int, valorRadio: double)
+establecerRadio(valorRadio: double)
+obtenerRadio(): double
+obtenerDiametro(): double
+obtenerCircunferencia(): double
+obtenerArea(): double
+obtenerNombre(): String
+toString(): String
PruebaHerenciaAbstracta
+main(args: String)
<abstract>Figura
+obtenerArea(): double
+obtenerVolumen(): double
+obtenerNombre(): String

12.7 Bibliografía

Booch, G. (1996). *Análisis y diseño orientado a objetos con aplicaciones*. Ciudad: México. Addison Wesley.

Deitel y Deitel (2012). *Cómo programar en Java*. Ciudad: México. Editorial Pearson (Prentice Hall).

Fowler, M. y Scott, K. (1999). *UML gota a gota*. Ciudad: México. Editorial Pearson.

Graig, L. (2003). *UML y patrones: una introducción al análisis y diseño orientado a objetos y al proceso unificado*. Segunda edición. Ciudad: México. Prentice Hall.

Rumbaugh, J., Jacobson, I. y Booch, G.: (2005). *The Unified Modeling Language User Guide*. Ciudad: San Francisco. Addison-Wesley, Reading, Mass. ISBN-13: 078-5342267976, ISBN-10: 0321267974.

Villalobos, J. y Casallas, R. (2006). *Fundamentos de programación, aprendizaje activo basado en casos*. Ciudad: México. Editorial Pearson.

Capítulo 13

Modelado orientado a objetos y aplicaciones de software

13.1 Temática a desarrollar

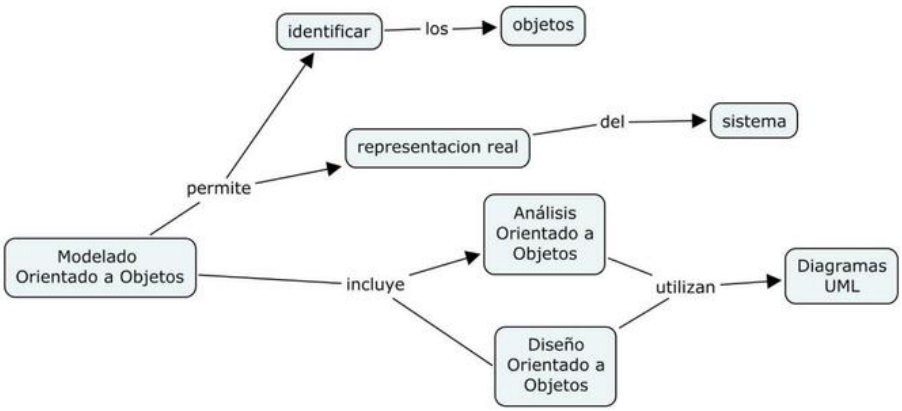
- Análisis orientado a objetos
- Diseño orientado a objetos
- Diagramas UML
- Aplicaciones de software

13.2 Introducción

El enfoque orientado a objetos brinda una serie de ventajas, como la familiaridad con los conceptos, facilitando de esta forma la comunicación con el usuario o cliente para representar sistemas sin determinar las características de implementación en el computador, permitiendo pasar directamente el dominio del problema al modelo del sistema en un proceso de modelación más natural, similar a como el hombre lo visualiza.

Este capítulo esboza los parámetros generales que se deben tener en cuenta para el buen desarrollo de una aplicación de software con enfoque orientado a objetos, empezando con el análisis, luego los requerimientos y los elementos relacionados con el diseño de la misma, para terminar con un ejemplo concreto como aplicación de software en Java.

Figura 27.
Mapa mental del modelado orientado a objetos



13.3 Análisis orientado a objetos

Es un método de análisis que examina los requisitos desde la perspectiva de las clases y objetos que se encuentran en el enunciado o términos del dominio del problema, tomando como punto de referencia entidades tangibles halladas en el problema; tales entidades varían dependiendo de los diversos casos prácticos, pero, en todos los casos, son elementos reales que forman parte del problema de forma directa.

El Análisis Orientado a Objetos (AOO) se basa en conceptos de objetos y atributos, el todo y las partes, clases y miembros. Este enfoque pretende conseguir modelos que se ajusten mejor al problema real, a partir de conocer el dominio del problema.

Los documentos que deben tenerse o desarrollarse durante el análisis son:

- Especificación de requisitos o requerimientos (definición y especificación)
- Escenarios y subescenarios
- Prototipos y su evaluación
- Diagramas de casos de uso

La especificación de requisitos o requerimientos comprende las tareas relacionadas con la determinación de las necesidades o de las condiciones a satisfacer para un software nuevo o modificado, tomando en cuenta los diversos requerimientos de los clientes. El propósito es hacer que los mismos alcancen un estado óptimo antes de obtener la fase de diseño en el proyecto. Los requerimientos deben ser medibles, comprobables, sin ambigüedades o contradicciones.

Para la especificación de requerimientos se debe:

- Identificar las necesidades del usuario (del negocio).
- Describir los objetivos de la aplicación.
- Definir características y funciones generales de la aplicación.
- El equipo de ingeniería de sistemas y los clientes deben establecer en conjunto las metas y objetivos de la aplicación.

13.4 Escenarios

Es una descripción parcial y concreta del comportamiento de un sistema en una determinada situación. Los escenarios son historias de usuarios y sus actividades. Los escenarios destacan objetivos sugeridos por la apariencia y comportamiento del sistema, refiriéndose a lo que los usuarios quieren hacer con el sistema, qué procedimientos se usan, cuáles no se usan, si se realizan o no satisfactoriamente y qué interpretaciones hacen de lo que les sucede.

Elementos de los escenarios:

- Configuración (*setting*). Por ejemplo, una oficina con un empleado enfrente del computador trabajando en una hoja electrónica. Es importante concretar que la persona es el auxiliar contable y que los objetos de trabajo son balances y presupuestos.
- Agentes o actores. El auxiliar contable es el único agente en el escenario que se describe, pero es normal que en las actividades humanas participe más de un actor, cada uno con sus objetivos.
- Cada escenario implica por lo menos un agente y al menos un objetivo. En el caso de que haya más de un actor y objetivo, se define un agente como actor principal.

13.5 Prototipo

Un prototipo puede ser un esquema en papel de una pantalla o pantallas, una imagen electrónica, una simulación en video de una tarea, una representación de una red de computadores, entre otros; de hecho, un prototipo puede ser cualquier cosa, desde una representación basada en papel hasta un artefacto o pieza de software.

Son características de los prototipos:

- Una representación de aquellos aspectos del software que serán visibles para el cliente (por ejemplo, la configuración de la interfaz con el usuario y el formato de los despliegues de salida).
- El prototipo es evaluado por el cliente para una retroalimentación; gracias a esta, se refinan los requisitos del software que se desarrollará.
- El prototipo permite brindar una representación limitada de un diseño a los usuarios para interactuar con él. Es una herramienta útil para hacer participar al usuario en el desarrollo y poder evaluar el producto ya en las primeras fases del desarrollo.

13.6 Diagramas de casos de uso

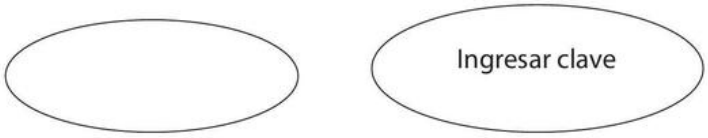
Los casos de uso son una técnica para capturar información de cómo un sistema o negocio funciona actualmente o de cómo se requiere que se desempeñe. Cada caso de uso proporciona uno o más escenarios que indican cómo debería interactuar el sistema con el usuario o con otro sistema para obtener un objetivo específico.

El modelo de casos de uso especifica la funcionalidad que el sistema debe ofrecer y está conformado por los siguientes elementos:

- Actores
- Casos de uso
- Relaciones

Actores: modelan diferentes roles que pueden representar los usuarios, máquinas, sensores, entre otros, que interactúan con el sistema a desarrollar. Se denota por una figura humana plana.

Casos de uso: representan todo lo que el usuario puede realizar con el sistema. Se denota mediante un óvalo, con la acción correspondiente.

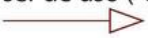


Relaciones: permiten asociar actores y casos de uso. Las relaciones que se pueden dar en un diagrama de casos de uso son:

- **Asociación.** Es el tipo de relación más básica que indica la invocación, desde un actor o caso de uso a otra operación (caso de uso). Esta relación se representa con una línea simple.

- **Dependencia o instanciación.** Es una forma muy particular de relación entre clases, en la cual una clase depende de otra, es decir, se instancia (se crea). Esta relación se representa con una flecha punteada.

- **Generalización**. Este tipo de relación es uno de los más utilizados y cumple una doble función, dependiendo de su estereotipo, que puede ser de uso ("uses") o de herencia ("extends").

- **Extends.** Se utiliza cuando un caso de uso es similar a otro en sus características.

- **Uses.** Utilizado cuando se tiene un conjunto de características que son similares en más de un caso de uso y no se desea mantener copiada la descripción de la característica.

Ejemplos:

Ejemplo 1

Se requiere construir un sistema de información que permita modelar las actividades que realiza un usuario con los procesos que hace en su correo electrónico: enviar y recibir mensajes electrónicos y actualizar direcciones de contactos, entre otras, teniendo en cuenta que se debe ingresar al sistema de correo con el *login* y el *password*.

Actor

usuario

Las principales actividades que realiza el usuario en su cuenta de correo son:

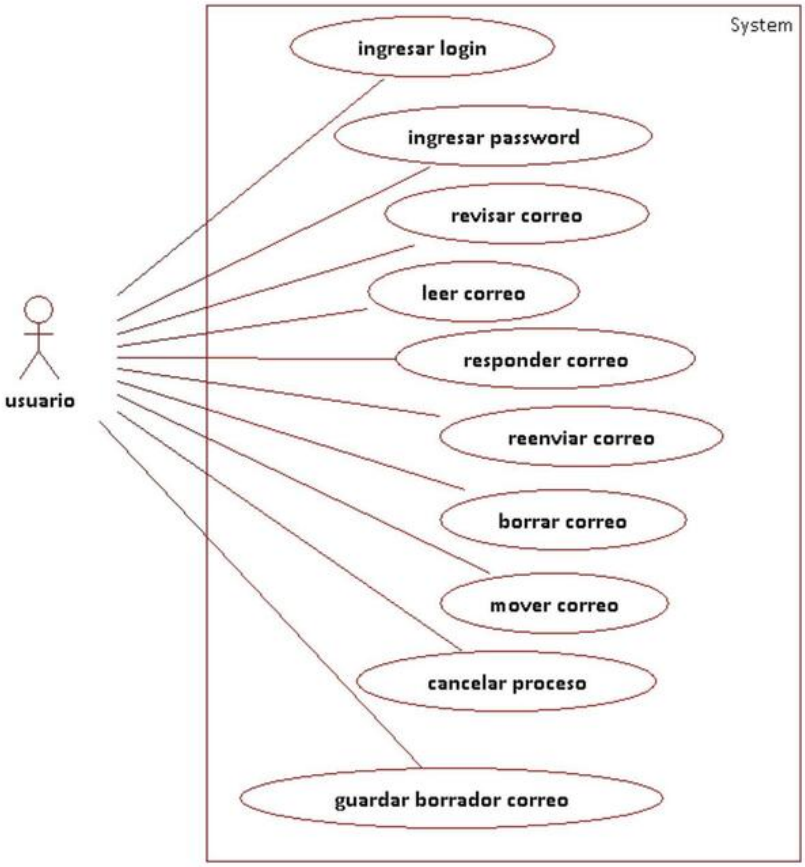
1. Ingresar *login* y *password*.
2. Revisar correos pendientes.
3. Leer correo.
4. Responder correo.
5. Reenviar comunicación a un determinado grupo.
6. Cancelar envío de correo.
7. Guardar correo en borrador.
8. Borrar correos.

El diagrama del caso de uso estará conformado por el actor, que en este caso es el usuario, y los casos de uso corresponden a las actividades realizadas en la cuenta de correo, un cuadro que encierra los casos de uso que corresponde al sistema.

Son características de los diagramas de casos de uso:

- Describen qué es lo que debe hacer el sistema, pero no cómo.
- No son diseñados para representar el diseño y no puede describir los elementos internos de un sistema.

- Sirven para facilitar la comunicación con los futuros usuarios del sistema y con el cliente y resultan especialmente útiles para determinar las características necesarias que tendrá el sistema.



Ejemplo 2

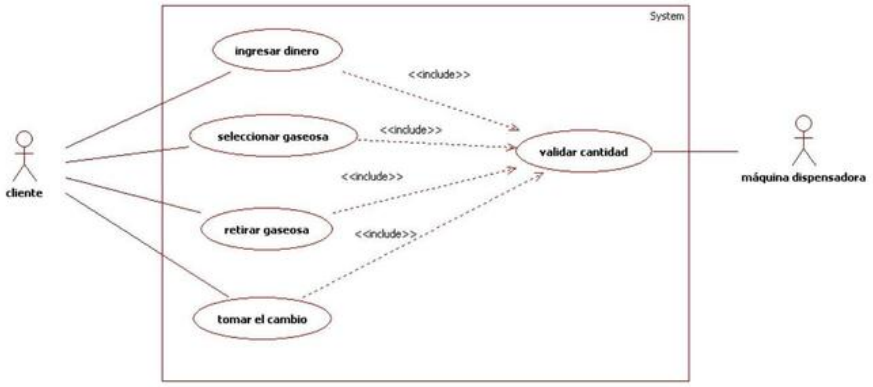
La administradora de la cafetería de la universidad requiere de una aplicación que le permita automatizar el proceso de compra de las gaseosas de una determinada marca para una máquina expendedora, donde cada gaseosa tiene un costo de $2.500, y para que la persona interesada pueda adquirir el producto debe ingresarle el valor en monedas, cuyas denominaciones pueden ser de $100, $200, $500; a continuación, el cliente podrá seleccionar la gaseosa,

donde todas tienen costos iguales disponibles en la máquina; además de eso, el sistema debe permitir retirar el producto cuando el valor ingresado sea mayor o igual a $2.500, el retiro de cambio se realiza para los casos en que haya ingresado un valor superior al costo del producto.

El actor es el cliente.

El nombre del caso de uso corresponde con el objetivo del actor, que es normalmente el que comienza el caso de uso, como, por ejemplo, ingresar dinero.

El diagrama de casos de uso de la máquina expendedora de gaseosas sería:



Para la documentación de los casos de uso se debe tener en cuenta:

- **Nombre del caso de uso:** se expresa mediante el uso de un verbo en infinitivo, por ejemplo, consultar información del cliente, imprimir documento, generar reporte, crear usuario.
- **Entradas:** datos de entrada proporcionados por el actor.
- **Resultados:** parcial o final, según sea el requerimiento.
- **Descripción:** resumen que expresa, de manera puntual, el requerimiento o la funcionalidad del sistema y el resultado que se espera que el sistema genere una vez ejecutada su función.
- **Precondición:** condición que describe en qué situación se debe encontrar el sistema y su entorno para poder comenzar el caso de uso.
- **Postcondición:** condición que describe en qué situación debe quedar el sistema y su entorno una vez que el caso de uso haya finalizado con éxito.

- **Flujo normal**: secuencia de interacciones entre los actores y el sistema que lleva a la finalización con éxito del caso de uso.
- **Flujo alternativo o excepciones:** situaciones eventuales y su tratamiento, que pueden darse durante la secuencia normal.

Los diagramas de casos de uso describen las relaciones y las dependencias entre un grupo de casos de uso y los actores participantes en el proceso. El siguiente formato presenta uno de los ítems de cómo podría ser una de las formas de representar la documentación de los casos de uso:

Nombre:	
Descripción:	
Entradas:	
Resultados:	
Actores:	
Precondiciones:	
Flujo normal:	
Actor	**Sistema**
Flujo alternativo:	
Postcondiciones:	

Según las necesidades, los casos de uso pueden especificarse con distinto detalle. Como un requisito funcional clásico: se especifica el servicio que debe proporcionar el sistema a los usuarios en texto libre. Por ejemplo:

Nombre:	R1 – Ingresar monedas a la máquina
Descripción:	Se requiere registrar el valor de las monedas depositadas en la máquina por parte del cliente.
Entradas:	
Moneda 100, Moneda 200, Moneda 500.	
Resultados:	
Total valor dinero depositado con base a las monedas que han sido ingresadas.	
Precondiciones:	
No existe valor procesado previo.	
Flujo normal:	
1. El usuario pulsa sobre el botón correspondiente al valor de la moneda. 2. El sistema muestra en una caja de texto el total del valor ingresado. 3. El sistema comprueba el valor de las monedas ingresadas y va generando el total acumulado. 4. El sistema deshabilita opciones cuando se ha ingresado el valor del producto o un valor superior.	
Flujo alternativo:	
El usuario no tiene el dinero suficiente. El sistema no entrega el producto hasta completar el valor de este, verifica el total ingresado y le avisa al usuario de ello permitiéndole que lo corrija.	
Postcondiciones:	
El valor total ha sido registrado en el sistema.	

Nombre:	R2 – Retirar gaseosa
Descripción:	Permitir reclamar producto.
Entradas:	
Solicitud producto.	
Resultados:	
Total valor del dinero depositado en base a las monedas que han sido ingresadas.	
Precondiciones:	
El usuario ingresó monedas.	
Flujo normal:	

1. El usuario pulsa sobre el botón correspondiente para solicitar producto. 2. El sistema verifica total. 3. El usuario toma el producto.
Flujo alternativo:
El sistema comprueba el total del dinero; si el valor es menor, le avisa al usuario de ello, permitiéndole que complete la cantidad.
Postcondiciones:
El usuario recibe el producto. El usuario recibe un mensaje de error.

13.7 Diseño orientado a objetos

Es una fase de la metodología orientada a objetos para el desarrollo de software. Su uso induce a los programadores a pensar en términos de objetos, en vez de procedimientos, cuando planifican su código. Un objeto agrupa datos encapsulados y procedimientos para representar una entidad.

El diseño orientado a objetos es la disciplina que define los objetos y sus interacciones para resolver un problema de negocio que fue identificado y documentado durante el análisis orientado a objetos.

- Tarjetas CRC (CRC: Clase, Responsabilidades, Colaboradores). Es una herramienta que por su sencillez permite centrarse en la importancia del objeto y sus operaciones en el sistema.

Como una extensión informal a UML, la técnica de las tarjetas CRC se puede usar para guiar el sistema a través de análisis guiados por la responsabilidad. Las clases se examinan, se filtran y se refinan con base en sus responsabilidades con respecto al sistema y las clases con las que necesitan colaborar para completar sus responsabilidades.

Una tarjeta CRC es una tarjeta indexada de 3 x 5 cm conformada por:

- El nombre de la clase y su descripción
- La responsabilidad de la clase que incluye:
- Conocimiento interno de la clase
- Servicios brindados por la clase

- Los colaboradores para las responsabilidades. Un colaborador es una clase cuyos servicios son necesarios para una responsabilidad.

Nombre de la abstracción (clase)	
Responsabilidades (operaciones y atributos)	**Colaboradores** (Relaciones de las clases) Elementos con los que va a interactuar la abstracción.

- Típicamente los sustantivos de los casos de uso o de los requerimientos del proyecto son útiles para encontrar candidatos a clases, correspondientes al dominio del problema.
- Los verbos de los casos de uso son candidatos a responsabilidades.

Enunciado técnico

La administradora de la cafetería de la universidad requiere de una aplicación que le permita automatizar el proceso de compra de las gaseosas de una determinada marca para una máquina expendedora, donde cada gaseosa tiene un costo de $2.500, y para que la persona interesada pueda adquirir el producto debe ingresarle el valor en monedas, cuyas denominaciones pueden ser de $100, $200, $500. A continuación, el cliente podrá seleccionar la gaseosa, donde todas tienen costos iguales disponibles en la máquina; además de eso, el sistema debe permitir retirar el producto cuando el valor ingresado sea mayor o igual a $2.500, la entrega de cambio se realiza para los casos en que haya ingresado un valor superior al costo del producto.

La máquina presentará mensajes de error cuando no haya existencias del producto solicitado, cuando falte ingresar más monedas para completar el valor de la gaseosa o cuando se intente retirar cambio cuando se ingresa el valor completo del producto.

Lista de palabras clave:

- Administradora
- Aplicación
- Permita automatizar
- Compra
- Gaseosas
- Productos
- Retirar producto
- Entregar cambio
- Ingresado
- Disponible
- Mensajes de error

- Costo
- Persona
- Usuario
- Producto
- Cambio
- Máquina
- Costo

Tarjetas CRC

Nombre de la clase: Usuario	
Responsabilidades • Ingresar monedas • Seleccionar producto • Retirar productos • Tomar el producto	**Colaboradores** • Máquina • Monedas

Nombre de la clase: Producto	
Responsabilidades • Mantener número de productos existentes	**Colaboradores** • Máquina • Usuario

Nombre de la clase: Moneda	
Responsabilidades • Contabilizar cantidad de monedas por denominación y sus correspondientes totales • Mostrar total de dinero ingresado	**Colaboradores** • Máquina

Nombre de la clase: Máquina	
• **Responsabilidades** • Entregar producto • Entregar cambio • Presentar total dinero • Presentar total cambio • Presentar mensaje de error	**Colaboradores** • Usuario • Monedas • Producto

Listado tentativo de clases:

- Denominaciones
- Producto
- Máquina
- Usuario

13.8 Diagramas UML (Unified Modeling Language)

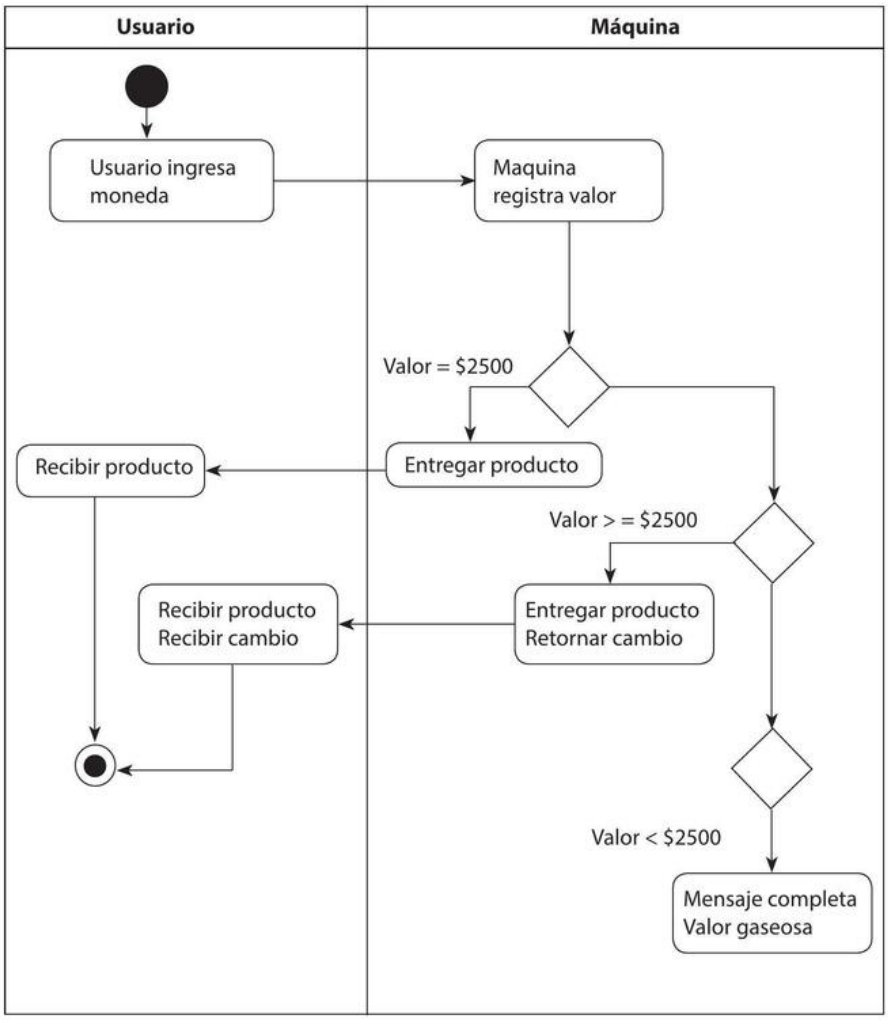
UML es un lenguaje que permite modelar, construir y documentar los elementos que forman un sistema software orientado a objetos. Se ha convertido en el estándar de la industria del software, debido a que ha sido impulsado por los autores de los tres métodos más usados de orientación a objetos: Grady Booch, Ivar Jacobson y Jim Rumbaugh.

El lenguaje UML nace de la necesidad de un lenguaje preciso de especificación que permitiera la comunicación entre los que diseñan el software y los que lo construyen.

Dentro de los tipos de diagramas a utilizar están:

- **Vista estructural:**
  - Diagrama de clases: muestra las clases y sus asociaciones.
  - Diagrama de objetos: corresponde a una instancia particular del Diagrama de clases.
- **Vista de usuario:**
  - Diagrama de casos de uso.
- **Vista de comportamiento:**
  - Diagrama de actividades.

Diagrama de actividades para la máquina de gaseosas del enunciado anterior:



13.9 Aplicaciones de software

El desarrollo de una aplicación de software en Java implica una serie de actividades que a continuación se relacionan:

- La identificación de los requerimientos del software a partir de las necesidades que se deben satisfacer del usuario o del negocio.

- Diseño de la aplicación de software: está conformado por los componentes de una aplicación (entidades del negocio) y describe en general el cómo se construirá una aplicación de software. Para ello, se hace uso de diagramas UML de clases, despliegue, secuencia, entre otros.
- Programación: construcción de la aplicación de software en un lenguaje de programación a partir de los diagramas elaborados en la etapa de diseño.
- Pruebas: hacen referencia a la comprobación de las necesidades que satisface el software, esto es, que responda correctamente a los requerimientos planteados en la especificación del problema.
- Documentación: corresponde a la documentación que es propia del desarrollo del software, dentro de la que están diferentes diagramas UML utilizados y manuales de usuario del sistema, entre otros.
- Mantenimiento: etapa en la que se realizan ajustes y mejoras a la aplicación de software construido.

El mapa mental de la figura 27 amplía el concepto de una aplicación de software.

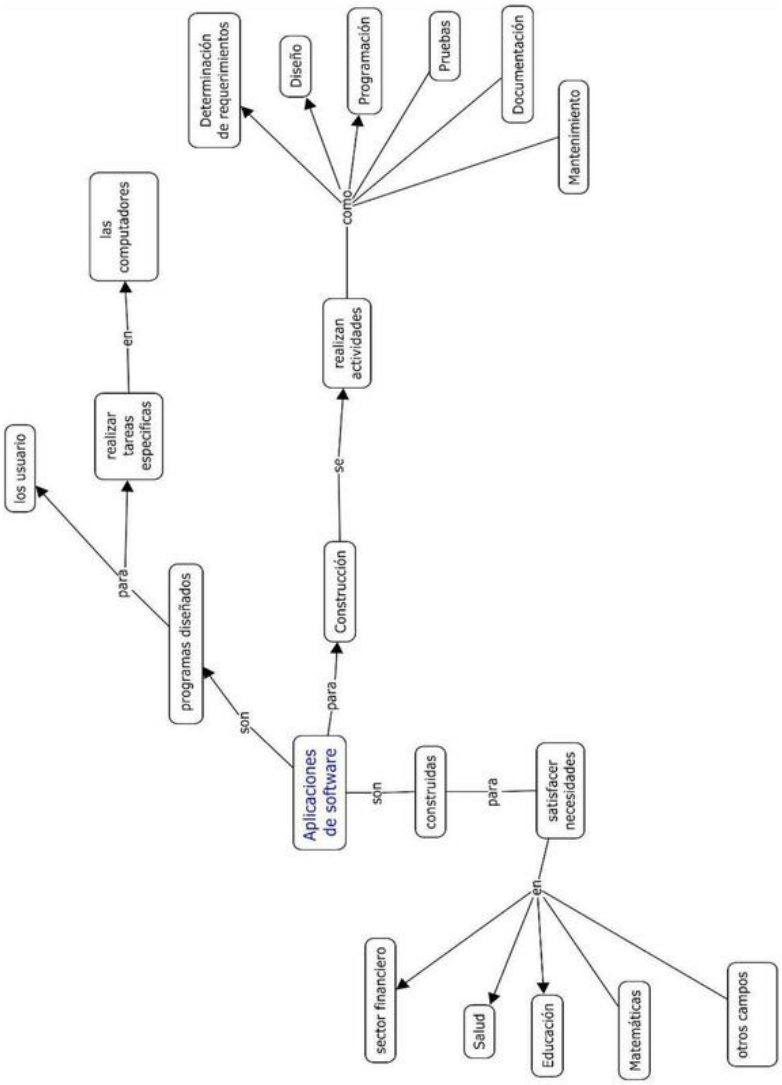


Figura 27.
Mapa mental de aplicaciones de software

Ejemplo:

El centro médico El Buen Samaritano, cuya misión es velar por el buen estado nutricional de sus pacientes, requiere la implementación de un aplicativo de software para que su personal médico pueda calcular el Índice de Masa Corporal (IMC) de cada paciente que es atendido. Para ello se apoya en la siguiente tabla desarrollada por la Organización Mundial de la Salud:

Clasificación	Clase	IMC (kg/m²)
Bajo peso		<18,50
	Delgadez severa	<16,00
	Delgadez moderada	16,00 - 16,99
	Delgadez ligera	17,00 - 18,49
Rango normal		18,50 - 24,99
Sobrepeso		≥25,00
	Preobeso	25,00 - 29,99
Obeso		≥30,00
	Obeso clase I	30,00 - 34-99
	Obeso clase II	35,00 - 39,99
	Obeso clase III	≥40,00

Fuente: Organización Mundial de la Salud.

Lo recomendado para un estado nutricional bueno es aquel valor personal que se encuentra dentro del rango específico como normal, en valores que van de 18,50 a 24,99. La Organización Mundial de la Salud (OMS) define el sobrepeso cuando el IMC es mayor o igual a 25 kg/m^2 y obesidad cuando el IMC es mayor o igual a 30 kg/m^2.

Los valores registrados en la anterior tabla son independientes de la edad y son para ambos sexos.

El IMC de una persona se define por la relación entre el peso y la estatura aplicando la siguiente fórmula:

$$\text{Índice de Masa Corporal (IMC, kg/m}^2\text{)} = \text{Peso(kg)} / \text{Estatura(m}^2\text{)}$$

A partir de la anterior información, se requiere la construcción e implementación de una aplicación de software que permita calcular:

- El índice de masa corporal expresado en:

  - Kilogramos y m2
  - Libras y pulgadas (unidades de medida)

- El estado nutricional de una persona

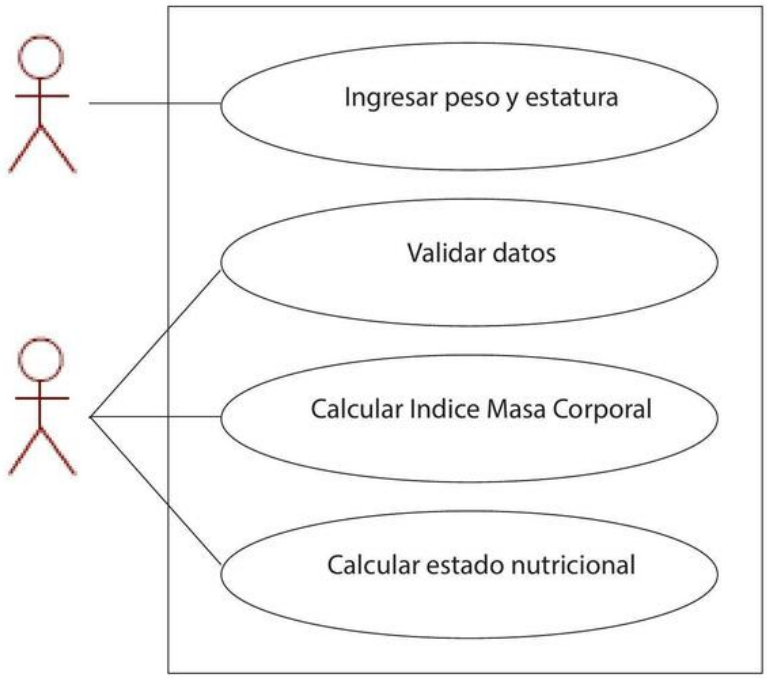
Actividades propuestas:

1. Determinar los requerimientos.
2. Elaborar el diagrama de casos de uso y documentarlos.
3. Elaborar el listado depurando sustantivos y verbos del enunciado del ejercicio.
4. Construir las tarjetas CRC.
5. Diseñar el diagrama de clases.
6. Construir la aplicación (codificación según diagramas de clases).
7. Generar la documentación (Javadoc) del programa.

a. Definición de requerimientos funcionales:

   - Ingresar información del paciente.
   - Validar datos.
   - Calcular el índice de masa corporal.
   - Determinar el estado nutricional.

b. Diagrama de casos de uso:



Documentación casos de uso:

Nombre:	(R1) Ingresar peso y estatura
Descripción:	Ingreso de datos correspondientes al peso y estatura como elementos para calcular el índice de masa corporal.
Entradas:	
Se dan a partir de las medidas del peso y la estatura del paciente y corresponden a datos numéricos positivos.	
Resultados:	
Registro de los datos correspondientes al peso y la estatura en el sistema.	
Actores:	
Usuario y el centro médico.	
Precondiciones:	
Ninguna.	
Flujo normal:	
Actor:	**Sistema:**

1. Usuario ingresa el pero y la estatura	2. Sistema visualiza peso y estatura. 3. Sistema registra datos ingresados.
Flujo alternativo:	
1.1. Usuario ingresa datos con valores negativos. 1.2. Usuario ingresa datos con formato errado.	
Postcondiciones:	
Sistema registra los datos ingresados.	

Documentación casos de uso:

Nombre:	(R2) Calcular Índice de Masa Corporal (IMC)
Descripción:	Se realiza proceso para obtener el IMC a partir del peso y la masa.
Entradas:	
Tomados del (R1) Ingresar peso y estatura.	
Resultados:	
Obtener el cálculo del IMC.	
Actores:	
Sistema (centro médico).	
Precondiciones:	
(R1)	
Flujo normal:	
Actor:	**Sistema:**
	1. Sistema calcula índice de masa corporal. 2. Visualización del resultado del índice de masa corporal.
Flujo alternativo:	
Ninguno	
Postcondiciones:	
Sistema tiene registrado el índice de masa corporal.	

Nombre:	(R3) Estado nutricional del paciente
Descripción:	Se realiza el proceso para obtener el estado nutricional del paciente del cálculo del IMC.
Entradas:	
Tomados del (R2) IMC calculado.	
Resultados:	
Obtener el estado nutricional del paciente.	
Actores:	
Sistema (centro médico).	
Precondiciones:	
(R2)	
Flujo normal:	
Actor:	**Sistema:**
	1. Sistema determina el estado nutricional del paciente. 2. Visualización del resultado del estado nutricional del paciente.
Flujo alternativo:	
Ninguno	
Postcondiciones:	
Sistema presenta el estado nutricional del paciente.	

Nota: desarrollar los otros casos de uso planteados en el ejercicio.

c. Enunciado del problema subrayando palabras clave.

El centro médico El Buen Samaritano, cuya misión es velar por el buen estado nutricional de sus pacientes, requiere la implementación de un aplicativo de software para que su personal médico pueda calcular el Índice de Masa Corporal (IMC) de cada paciente que es atendido.

Para ello se apoya en la siguiente tabla desarrollada por la Organización Mundial de la Salud.

Lo recomendado para un estado nutricional bueno es aquel valor personal que se encuentra dentro del rango específico como normal, en valores que van de 18,50 a 24,99. La Organización Mundial de la Salud (OMS) define el sobrepeso

cuando el IMC es mayor o igual a 25 kg/m^2 y obesidad cuando el IMC es mayor o igual a 30 kg/m^2.

Índice de Masa Corporal (IMC, kg/m^2) = Peso (kg) / Estatura (m^2)

A partir de la anterior información, se requiere la construcción e implementación de una aplicación de software que permita calcular:

- El índice de masa corporal expresado en:
  - o Kilogramos y m^2
  - o Libras y pulgadas (unidades de medida)
- El estado nutricional de una persona

Listado de palabras clave del enunciado:

Centro médico, paciente, médico, estado, aplicativo, índice de masa corporal, rango, normal, kilogramos y metros, libras y pulgadas

Listado de posibles clases del sistema:

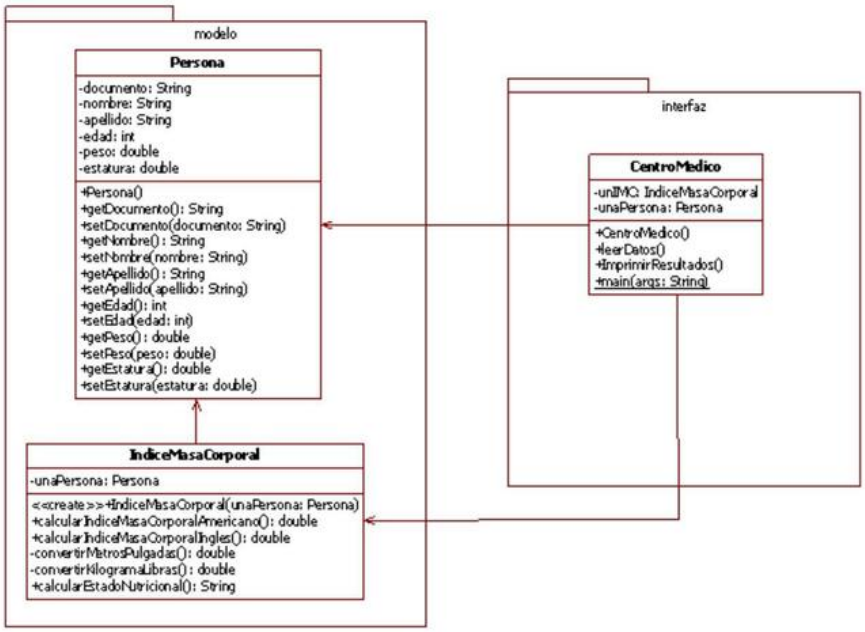
- Índice de masa corporal
- Centro médico

d. Elaboración de las tarjetas CRC:

Nombre de la clase: IndiceMasaCorporal	
Responsabilidad	**Colaboración**
Registrar información del peso y estatura del paciente.	Centro médico
Calcular el índice de masa corporal del paciente.	
Hallar estado nutricional del paciente.	

Nombre de la clase: CentroMedico	
Responsabilidad	**Colaboración**
Lectura y visualización de información.	IndiceMasaCorporal
Validar datos ingreso de datos numéricos.	

e. Diagrama de clases:



Documentación métodos de la clase IndiceMasaCorporal

Método para calcular el índice masa corporal expresado en kilogramos y metros (sistema americano).

Análisis:
Entrada: corresponde a los atributos de la clase de la masa y la estatura que fueron ingresados.
Salida: es el resultado de aplicar la fórmula:
Índice de Masa Corporal (IMC, kg/m^2) = Peso (kg) / Estatura (m^2).

Nombre del método	calcularIndiceMasaCorporalAmericano
Entrada (lista de parámetros)	No tiene
Resultado (tipo de dato de retorno)	double Índice de masa corporal
Solución planteada	Cálculo a partir aplicar la fórmula IMC = peso / estatura*estatura;
Firma del método	calcularIndiceMasaCorporalAmericano ( )

```
public double calcularIndiceMasaCorporalAmericano( ) {
        return  peso / estatura*estatura;
}
```

Pruebas de escritorio:

Supongamos las siguientes medidas:

Peso	Estatura	Retorno
60kg	1.72	20.28
78Kg	1.70	26.98

Documentación métodos de la clase IndiceMasaCorporal

Método calcularEstadoNutricional.

Análisis:
Entrada: corresponde a los atributos de la clase de la masa y la estatura que fueron ingresados.
Salida: es el resultado de aplicar la fórmula:

Índice de Masa Corporal (IMC, kg/m^2) = Peso (kg) / Estatura (m^2)

Nombre del método	calcularEstadoNutricional
Entrada (lista de parámetros)	No tiene
Resultado (tipo de dato de retorno)	double Índice de masa corporal

Solución planteada	A partir del índice de masa corporal, se evalúa si el peso está entre un determinado rango: si (imc < 16) si (imc >=16 && imc < 17) si (imc >=17 && imc < 18.5) si (imc >=18.5 && imc < 25) si (imc >=25 && imc < 30) si(imc >=30 && imc < 35) si (imc >=35 && imc < 40) si (imc >=40)
Firma del método	calcularEstadoNutricional( )

```
public String calcularEstadoNutricional( ) {
        double imc = calcularIndiceMasaCorporalAmericano();
        String estado = "";
    if (imc < 16) {
      estado = "Delgadez severa          ";
    } else if (imc >=16 && imc < 17) {
      estado = "Delgadez moderada";
    } else if (imc >=17 && imc < 18.5) {
      estado = "Delgadez ligera";
    } else if (imc >=18.5 && imc < 25) {
     estado = "Peso Normal";
    } else if (imc >=25 && imc < 30) {
      estado = "Pre-Obeso";
    } else if (imc >=30 && imc < 35) {
      estado = "Obeso Clase I";
    } else if (imc >=35 && imc < 40) {
      estado = "Obeso Clase II";
    } else if (imc >=40 ) {
      estado = "Obeso Clase III";
    }
    return estado;
}
```

Pruebas de escritorio:

Supongamos las siguientes medidas:

Peso	Estatura	IMC	Estado
60 kg	1.72	20.28	Peso normal
78 kg	1.70	26.98	Peso normal

f. Codificación de las clases IndiceMasaCorporal y CentroMedico:

```
/**
* clase que modela la información de la persona.
 *  @ Project : indice de masa corporal
 *  @ File Name : IndiceMasaCorporal
 *  @author : Miguel Hernández
 *  @version 1.0
 */

public class Persona {
        private String documento;
        private String nombre;
        private String apellido;
        private int edad;
        private double peso;
        private double estatura;

        public Persona() {
                this.documento = "123";
                this.nombre = "Luis Eduardo";
                this.apellido = "Baquero Rey";
                this.edad = 25;
                this.peso = 85;
                this.estatura = 190;
        }


        public Persona(String documento, String nombre, String apellido,
int edad, double peso, double estatura) {
```

```
        this.documento = documento;
        this.nombre = nombre;
        this.apellido = apellido;
        this.edad = edad;
        this.peso = peso;
        this.estatura = estatura;
}



public String getDocumento() {
        return documento;
}

public void setDocumento(String documento) {
        this.documento = documento;
}

public String getNombre() {
        return nombre;
}

public void setNombre(String nombre) {
        this.nombre = nombre;
}

public String getApellido() {
        return apellido;
}

public void setApellido(String apellido) {
        this.apellido = apellido;
}

public int getEdad() {
        return edad;
}

public void setEdad(int edad) {
        this.edad = edad;
}
```

```
	public double getPeso() {
		return peso;
	}

	public void setPeso(double peso) {
		this.peso = peso;
	}

	public double getEstatura() {
		return estatura;
	}

	public void setEstatura(double estatura) {
		this.estatura = estatura;
	}
}

/**
 * clase que modela las operaciones para calcular
 * índice de masa corporal y el estado nutricional
 *  @ Project : índice de masa corporal
 *  @ File Name : IndiceMasaCorporal
 *  @author : mhernandezb lebaqueror
 *  @version 1.0
 */
public class IndiceMasaCorporal {

	private Persona unaPersona;

	public IndiceMasaCorporal(Persona unaPersona) {
		this.unaPersona = unaPersona;
	}


	public double calcularIndiceMasaCorporalAmericano()
	{
		return   unaPersona.getPeso()/ ( unaPersona.getEstatura()*u-
naPersona.getEstatura());
	}

	public double calcularIndiceMasaCorporalIngles()
```

```
        {
                return  convertirMetrosPulgadas ( ) / (
convertirKilogramaLibras( ) *
convertirKilogramaLibras( ) );
        }

        private double convertirMetrosPulgadas ( ) {
                final double CONVERSOR = 39.37;
                return unaPersona.getEstatura() * CONVERSOR;
        }

        private double convertirKilogramaLibras( ) {
                final double CONVERSOR = 2.2046;
                return unaPersona.getPeso() * CONVERSOR;
        }

        public String calcularEstadoNutricional( ) {
                /*
    * Clasificación       Clase    IMC (Kg/m2)
    *   Bajo Peso                                                <18,50
    *
    *         Delgadez severa                    <16,00
    *         Delgadez moderada                  16,00 - 16,99
    *         Delgadez ligera              17,00 - 18,49
    *         Rango normal                       18,50 - 24,99
    *  Sobrepeso                                                 >=25,00
    *                    Pre-Obeso                        25,00 - 29,99
    *  Obeso                                                     >=30,00
    *                    Obeso Clase I              30,00 - 34-99
    *                    Obeso Clase II             35,00 - 39,99
    *                    Obeso Clase III            >=40,00
    */
    double imc = calcularIndiceMasaCorporalAmericano();
    String estado = "";
   if (imc < 16) {
      estado = "Delgadez severa          ";
   } else if (imc >=16 && imc <17) {
      estado = "Delgadez moderada";
   } else if (imc >=17 && imc <18.5) {
      estado = "Delgadez ligera";
   } else if (imc >=18.5 && imc <25) {
```

```
        estado = "Peso Normal";
      } else if (imc >=25 && imc <30) {
        estado = "Pre-Obeso";
      } else if (imc >=30 && imc <35) {
        estado = "Obeso Clase I";
      } else if (imc >=35 && imc <40) {
        estado = "Obeso Clase II";
      } else if (imc >=40 ) {
        estado = "Obeso Clase III";
      }
      return estado;
    }
}

/**
 * clase que modela los procesos de lectura e impresión
 * para calcular índice de masa corporal y el estado nutricional
 *  @ Project : índice de masa corporal
 *  @ File Name : CentroMedico
 *  @author : mhernandezb lebaqueror
 *  @version 1.0
 */
import java.text.DecimalFormat;
import java.util.Scanner;

public class CentroMedico {
        private IndiceMasaCorporal unIMC;
        private Persona unaPersona;

        public CentroMedico() {
                this.unaPersona = new Persona();
        }


        public void leerDatos() {
                String documento = "";
                String nombre = "";
                String apellido = "";
                int edad = 0;
                double peso = 0.0;
                double estatura = 0.0;
```

```
                Scanner lea = new Scanner(System.in);
                System.out.print("Documento ");
                documento = lea.nextLine();
                System.out.print("Nombre ");
                nombre = lea.nextLine();
                System.out.print("Apellido ");
                apellido = lea.nextLine();
                System.out.print("Edad ");
                edad = lea.nextInt();
                System.out.print("Estatura ");
                estatura = lea.nextDouble();
                System.out.print("Peso ");
        peso = lea.nextDouble();
                //cargar los datos al objeto
                unaPersona.setDocumento(documento);
                unaPersona.setNombre(nombre);
                unaPersona.setApellido(apellido);
                unaPersona.setEdad(edad);
                unaPersona.setEstatura(estatura);
                unaPersona.setPeso(peso);
                this.unIMC = new IndiceMasaCorporal(unaPersona);
        }

        public void ImprimirResultados( ) {
                //formato decimal de dos cifras
                DecimalFormat formatoDecimal = new
DecimalFormat("####.##");
                System.out.println("\nEl estado de la Persona\n");
                System.out.println("Documento "+unaPersona.
getDocumento());
                System.out.println("Nombre "+unaPersona.getNombre());
                System.out.println("Apellido "+unaPersona.getApellido());
                System.out.println("Edad "+unaPersona.getEdad());
                System.out.println("Estatura "+unaPersona.getEstatura());
                System.out.println("Peso "+unaPersona.getPeso());

                System.out.println("Indice de Masa Corporal en Kg/M2
"+formatoDecimal.format(unIMC.calcularIndiceMasaCorporalAmericano()));
                System.out.println("Indice de Masa Corporal lbs / pulg2
"+formatoDecimal.format(unIMC.calcularIndiceMasaCorporalIngles()));
                System.out.println("Estado Nutricional
```

```
"+unIMC.calcularEstadoNutricional());
        }

        public static void main(String[] args) {
                CentroMedico unCentroMedio = new CentroMedico();
                unCentroMedio.leerDatos();
                unCentroMedio.ImprimirResultados();

        }

}
```

g. Mediante la utilización de Eclipse o NetBeans, generar la respectiva documentación con la utilidad de Javadoc.

13.10 Lecturas recomendadas

- UML.
- Diagramas de UML.
- Principios del modelado.

13.11 Preguntas de revisión de conceptos

- Importancia del modelamiento de una aplicación de software.
- Qué es el modelado orientado a objetos.
- Utilidad de documentar una aplicación de software.

13.12 Ejercicios

Para cada uno de los siguientes enunciados, realizar las siguientes actividades.

a. Determinar requerimientos.
b. Elaborar diagrama de casos de uso y documentarlos.
c. Elaborar listado depurando sustantivos y verbos del enunciado del ejercicio.
d. Construir las tarjetas CRC.
e. Diseñar diagrama de clases.
f. Construir aplicación (codificación según diagramas de clases).

g. Generar la documentación (Javadoc) del programa.

1. El centro médico El Buen Samaritano, cuya misión es velar por la salud y bienestar de sus pacientes, en la actualidad, dentro de sus políticas para brindar un mejor servicio, necesita la implementación de un sistema automatizado, tomando como base los resultados obtenidos en el laboratorio de análisis clínicos, para que un médico determine si una persona tiene anemia o no, lo cual depende de su nivel de hemoglobina en la sangre, de su edad y de su género. Si el nivel de hemoglobina que tiene una persona está en el rango que le corresponde, este determina su resultado positivo y, en caso contrario, negativo. La información en la que el médico se basa para obtener el resultado es la siguiente:

Edad (meses)	Nivel hemoglobina	Género
0 - 1	13 - 26 g%	Femenino o masculino
1< x <= 6	10 - 18 g%	Femenino o masculino
6 < x <= 12	11 - 15 g%	Femenino o masculino

Edad (años)	Nivel hemoglobina	Género
1 < x < = 5	11.5 - 15 g%	Femenino o masculino
5 < x < = 10	12.6 - 15.5 g%	Femenino o masculino
10 < x < = 15	13 - 15.5 g%	Femenino o masculino
Mujeres >= 15	12 - 16 g%	Femenino
Hombres >= 15	14 - 18 g%	Masculino

2. Desarrollar una aplicación que permita realizar el modelado de la pantalla de un reloj digital e imprima las horas y minutos separadas por dos puntos, por ejemplo, 02:35; 12:40; 1:53 p. m.; las 11:59 a. m. es un minuto antes de las doce (00:00).
   La aplicación debe permitir adelantar o atrasar los minutos y las horas, fijar una hora inicial en el reloj e imprimir, además de las horas, los minutos y el meridiano (a. m., p. m.).

3. El departamento de ciencias básicas requiere una aplicación que permita al usuario seleccionar una de las siguientes figuras geométricas: la circunferencia, el cuadrado, el rectángulo, el rombo, el trapecio y el romboide, para luego calcular su área y perímetro.

La aplicación deberá:

- Permitir leer los datos por consola según sea el tipo de figura y solicitar únicamente los datos necesarios.
- Antes de imprimir el resultado, se debe verificar si los datos ingresados son valores positivos o, de lo contrario, volver a capturarlos.
- Imprimir los resultados de la figura seleccionada.

4. Se requiere construir una aplicación que permita digitar las credenciales para ingresar al sistema; tenga en cuenta las siguientes condiciones:

   - Las credenciales están conformadas por lo atributos login o usuario y la clave o contraseña. Por defecto, la longitud será de 12 caracteres.
   - La aplicación implementará los siguientes constructores:
     - ✓ Un constructor por defecto.
     - ✓ Un constructor que recibe como parámetro el tamaño de la contraseña, la cual se debe generar de forma aleatoria con esa longitud establecida.
     - ✓ Para efectos de cumplir con los requerimientos, se deben implementar los siguientes métodos:
       - **esFuerte()**: método booleano que retorna true o false según sea el caso; para que sea fuerte, debe tener más de 2 mayúsculas, más de 1 minúscula, más de 3 números y un símbolo especial.
       - **generarPassword()**: método que genera la contraseña aleatoria del objeto con la longitud que tenga.
       - Método obtener la contraseña y longitud.
       - Método para obtener el usuario.
       - Métodos para actualizar usuarios y contraseña.

13.13 Bibliografía

Aguilar, L. J. (2004). *Fundamentos de programación algoritmos y estructura de datos.* Tercera edición. Ciudad: México. McGraw-Hill.

Barnes, D. y Kölling, M. (2007). *Programación orientada a objetos con Java.* Ciudad: Madrid. Prentice Hall.

Bertrand, M. (1999). *Construcción de software orientado a objetos.* Segunda edición. Ciudad: Madrid. Prentice Hall.

Booch, G. (1996). *Análisis y diseño orientado a objetos con aplicaciones.* Ciudad: México. Addison Wesley.

Deitel y Deitel (2012). *Cómo programar en Java*. Ciudad: México. Editorial Pearson (Prentice Hall).

Fowler, M. y Scott, K. (1999). *UML gota a gota*. Ciudad: México. Editorial Pearson.

Graig, L. (2003). *UML y patrones: una introducción al análisis y diseño orientado a objetos y al proceso unificado*. Segunda edición. Ciudad: México. Prentice Hall.

Rumbaugh, J., Jacobson, I. y Booch, G. (2005). *The Unified Modeling Language User Guide*. Ciudad: San Francisco. Addison-Wesley, Reading, Mass. ISBN-13: 078-5342267976, ISBN-10: 0321267974.

Villalobos, J. y Casallas, R. (2006). *Fundamentos de programación, aprendizaje activo basado en casos*. Ciudad: México. Editorial Pearson.

Capítulo 14

Documentación

14.1 Temática a desarrollar

- Javadoc
- Soporte Javadoc para los IDE
- API Java 8

14.2 Introducción

La documentación de Java es muy completa para cada versión del JDK; esa documentación se puede consultar en línea o tenerla en un archivo aparte, previamente descargado. El API presenta los *packages*, clases e interfaces con descripciones bastante detalladas de las variables y métodos, así como sobre las relaciones jerárquicas.

La documentación de Java está escrita en HTML y se explora con un navegador web, como Microsoft Internet Explorer, Mozilla Firefox, Google Chrome, Opera o Safari.

14.3 Javadoc

Es una herramienta para generar documentación para una aplicación en Java a partir de comentarios. Javadoc realza algunos comentarios, de los que exige una sintaxis especial: deben comenzar por "/**" y terminar por "*/", incluyendo una descripción y algunas etiquetas especiales que se insertan en el código fuente de una aplicación en Java. La documentación se genera en formato HTML, la cual permite ser consultada por medio de un navegador.

El paquete de desarrollo de Java incluye la herramienta Javadoc, con la cual se genera un conjunto de páginas web a partir de los archivos de código correspondientes a los diferentes programas que conforman la aplicación. Esta herramienta toma en consideración algunos comentarios para generar una documentación estructurada de las clases, atributos y métodos.

Es de anotar que Javadoc ayuda a la comprensión de la arquitectura de la solución; se dice que Javadoc se centra en la interfaz (API - *Application Programming Interface*) de las clases y paquetes de Java.

La herramienta Javadoc

La sintaxis con la que tiene que ser invocada la herramienta es: javadoc [opciones] Clase.

Aunque existe una gran variedad de opciones, estas son algunas de las que nos pueden resultar útiles en el desarrollo de las prácticas:

- javadoc-public Clase: presenta solo las clases y los miembros públicos.
- javadoc-protected Clase: presenta las clases y los miembros protegidos.
- javadoc-package Clase: presenta las clases y los miembros con acceso de paquete, los protegidos y los públicos.
- javadoc-private Clase: presenta todas las clases y todos los miembros, incluyendo los privados.

Las etiquetas

Aparte de los comentarios propios, la utilidad Javadoc proporciona una serie de etiquetas para completar la información que se necesite dar de una determinada clase o método.

La herramienta Javadoc escanea las etiquetas cuando están incluidas dentro del programa en un comentario Javadoc.

Estas etiquetas para documentación permiten generar una API en HTML a partir del código fuente con los comentarios. Las etiquetas comienzan con el símbolo arroba (@) y son sensibles a mayúsculas y minúsculas.

Una etiqueta se debe ubicar al inicio de una línea o, al menos, solo precedida por espacio(s) y asterisco(s), pues de lo contrario será tratada como texto normal por el compilador.

Hay dos tipos de etiquetas:

- Etiquetas de bloque: solo se pueden utilizar en la sección de etiquetas que sigue a la descripción principal. Son de la forma: @etiqueta
- Etiquetas inline: se pueden utilizar tanto en la descripción principal como en la sección de etiquetas. Son de la forma: {@tag}, es decir, se escriben entre los símbolos de llaves.

Ejemplos de algunas etiquetas:

@author

Muestra el autor de la clase en el argumento nombre. Un comentario de este tipo puede tener más de un autor, en cuyo caso se pueden usar tantas etiquetas de este tipo como autores hayan colaborado en la creación del código fuente o bien se puede poner a todos en una sola etiqueta. En este último caso, Javadoc inserta una coma (,) y un espacio entre los diferentes nombres.

```
/**
 * @author (your name)
 */
/**
 * @author Luis Eduardo Baquero, Miguel Hernández Bejarano
 */
```

@deprecated

Esta etiqueta marca un identificador como desactualizado, es decir, inadecuado para continuar con su uso. Es conveniente asegurarse de que la entidad desactualizada continúa funcionando de forma que no se dañe el código existente que no ha sido actualizado. La desactualización ayuda a animar a los desarrolladores de código a actualizarse a la última versión, preservando la integridad del código existente.

Ejemplos:

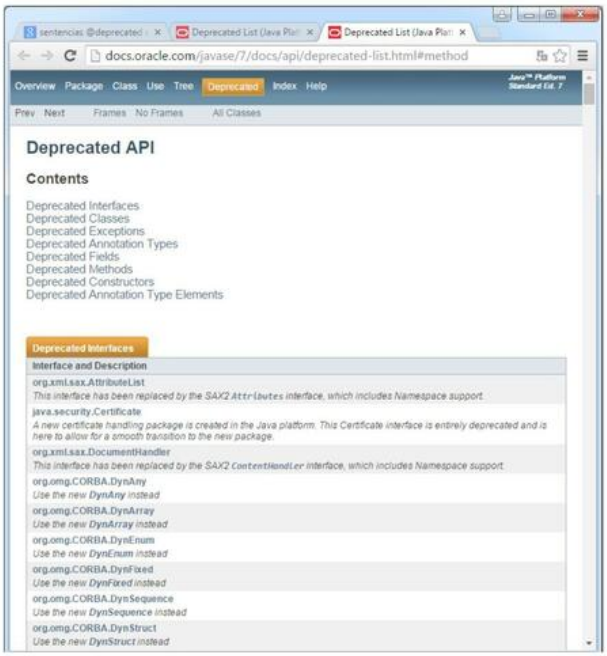
@Deprecated Este método se eliminará en próximas versiones

```
public double obtenerVolumen() {
  return 0.0;
}
```

@Deprecated: para marcar un método como desaprobado, no debe ser utilizado en la programación, ya que de seguro puede generar problemas, aunque esté disponible.

```
@Deprecated
public static void métodoEstatico(){
  System.out.println("Estático Desaprobado");
}
```

Para obtener mayor información al respecto, esta el API de Java, tal como se presenta en la siguiente imagen del sitio oficial http://docs.oracle.com/javase/7/docs/api/deprecated-list.html#method



{@link}

Se debe crear un enlace con el texto "etiqueta" que apunta a la documentación del paquete, la clase o el miembro especificado.

```
/** Método que invoca a {@link #obtenerLectura(String)}.  */

 public void processLineByLine() {
          obtenerLectura(scanner.nextLine());

  }

 protected void obtenerLectura(String aLine){
           Scanner scanner = new Scanner(aLine);

 }
```

@param

Anexa un parámetro en la sección "Parámetros". La descripción se puede escribir en más de una línea. Esta etiqueta solo es válida en los comentarios para documentación de métodos y constructores.

```
/**
 * constructor con dos parámetros
 * @param x
 * @param y
 */

 public Punto (double x, double y) {
          this.x = x;
          this.y = y;
 }
  /**
```

```
 * método que actualiza el apellido
 * @param apellido
 */
        public void setApellido(String apellido) {
                this.apellido = apellido;
        }
```

@return

Agrega una cabecera con un enlace que apunta a una referencia. Un comentario para documentación puede contener más de una de estas etiquetas; todos los enlaces de este tipo se agrupan bajo la misma cabecera.

```
/**
        * Método que retorna el objeto de la fecha
        */
        public String toString() {
                return ""+dia + "/" + mes + "/" + año;
        }

/**
        * método que retorna el apellido
        * @return nombre
        */
        public String getApellido() {
                return apellido;
        }
```

@version

Anexa en la cabecera de la documentación generada con la versión de esta clase. Por versión, normalmente se hace referencia a la versión de la aplicación de software.

Ejemplo:

```
/**
 * @version 1.0
```

```
 */
/**
 * @version 24 de julio de 2015
 */
```

14.4 Javadoc en los IDE

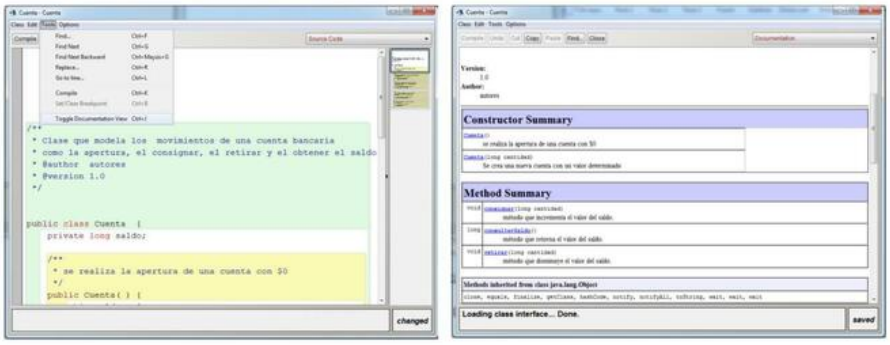
BlueJ

Soporte de BlueJ para Javadoc. Si un proyecto ha sido comentado haciendo uso de Javadoc, BlueJ ofrece utilidades para generar la documentación HTML completa, para lo cual, en la ventana principal, se debe seleccionar la opción "Tools/Project Documentation" del menú; luego se generará la documentación y se visualizará en la ventana de un navegador.

En el editor de BlueJ, se puede pasar de la vista del código fuente de una clase a la vista de su documentación cambiando la opción "Source Code" a "Documentation" en la parte derecha de la ventana del menú "Tools" del editor. Esta opción ofrece una vista previa y rápida de la documentación, pero no contendrá referencias a la documentación de las superclases o de las clases que se usan.

La figura 28 presenta la documentación en BlueJ.

Figura 28.
Documentación en BlueJ



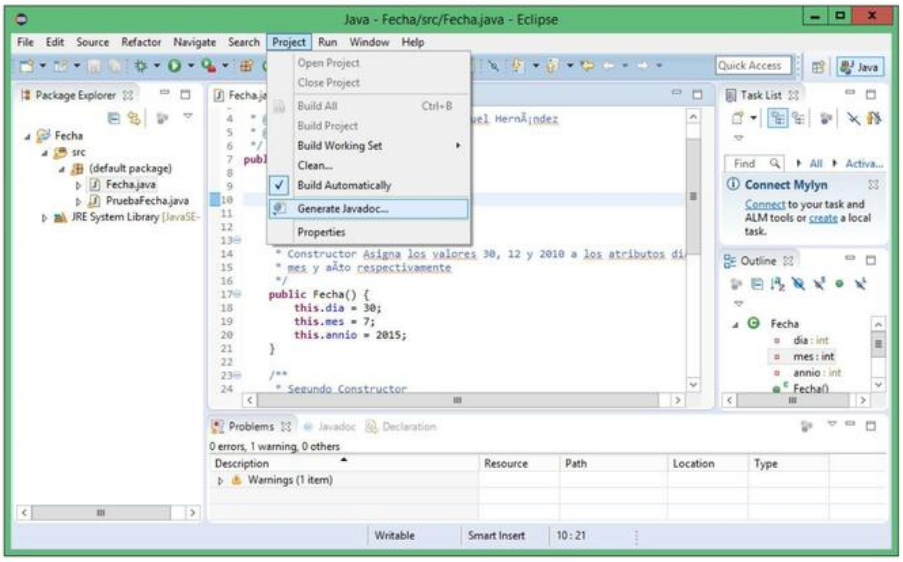
Documentación en Java **Documentación Javadoc**

Soporte de Javadoc para Eclipse

Eclipse permite, de una forma muy sencilla, generar, automáticamente, la documentación del propio proyecto.

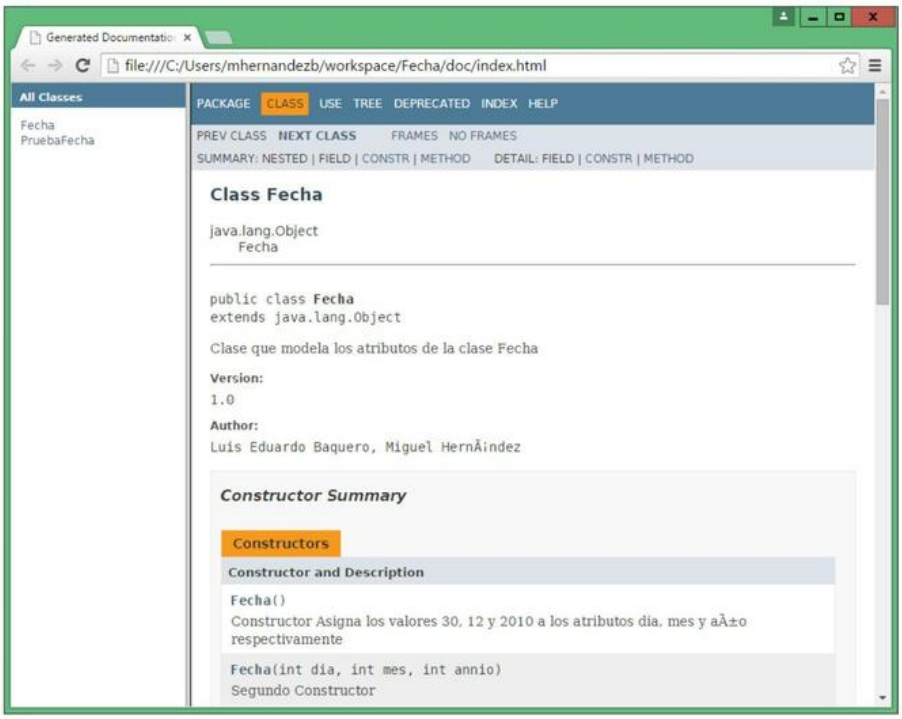
Antes de crear los archivos de documentación, es necesario configurar la herramienta Javadoc que Eclipse debe utilizar. Para ello, basta con escribir la ubicación del ejecutable Javadoc en la opción "Window-Preferences-Java-Javadoc" accesible desde el menú principal.

Figura 29.
Generación Javadoc en Eclipse



En la figura 30 se visualiza la documetación Javadoc generada en Eclipse.

Figura 30.
Documentación Javadoc en Eclipse



Soporte de Javadoc para Netbeans

NetBeans dispone de todo un entorno para crear documentación Javadoc a partir de "Tools-Javadoc Manager", que es como se llama al administrador de Javadoc, el cual es capaz de montar filesystems que contienen documentos Javadoc y gestionar su uso. Lo normal es que exista una carpeta llamada "docs", dentro de la raíz del paquete, que es en la cual se genera la documentación.

En la carpeta de documentación, el archivo index.html será el índice general. Mediante el administrador de archivos, se debe montar esta carpeta (mount) para que pueda ser buscada.

Figura 31.
Generación Javadoc en Netbeans

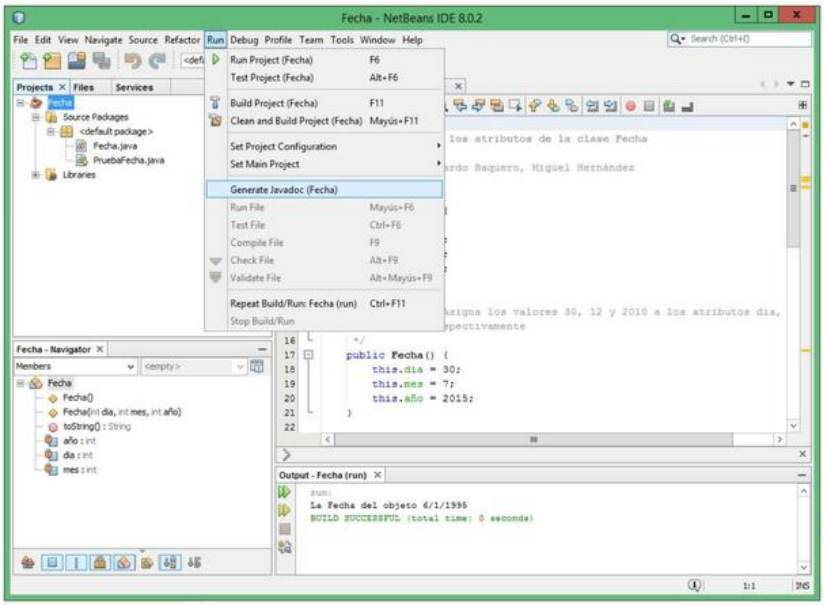
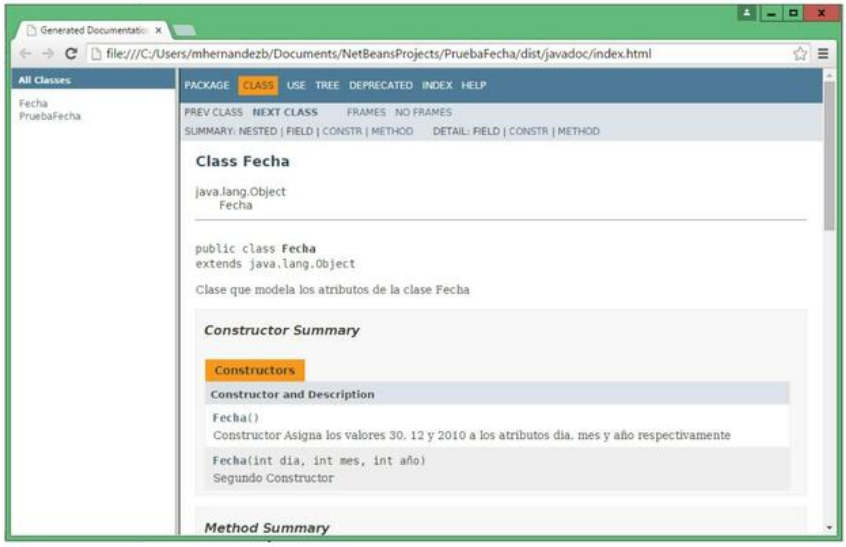


Figura 32.
Documentación Javadoc en Netbeans



14.5 API de Java

Es una herramienta que integra y provee de un conjunto de clases utilitarias para la construcción de aplicaciones de software en Java.

El API incluye las clases de la versión correspondiente (6, 7, 8), con sus correspondientes métodos constructores y demás métodos asociados a cada clase.

El API está organizado en paquetes lógicos, donde cada paquete contiene un conjunto de clases relacionadas semánticamente, como, por ejemplo:

- java.io: entrada y salida de datos.
- java.util: estructuras de datos implementadas para que sean usadas.

El API de Java es una herramienta de apoyo de consulta para los desarrolladores de aplicaciones en Java.

En la figura 33 *(véase en página siguiente)* se presenta, en el API de Java 8, la clase Random.

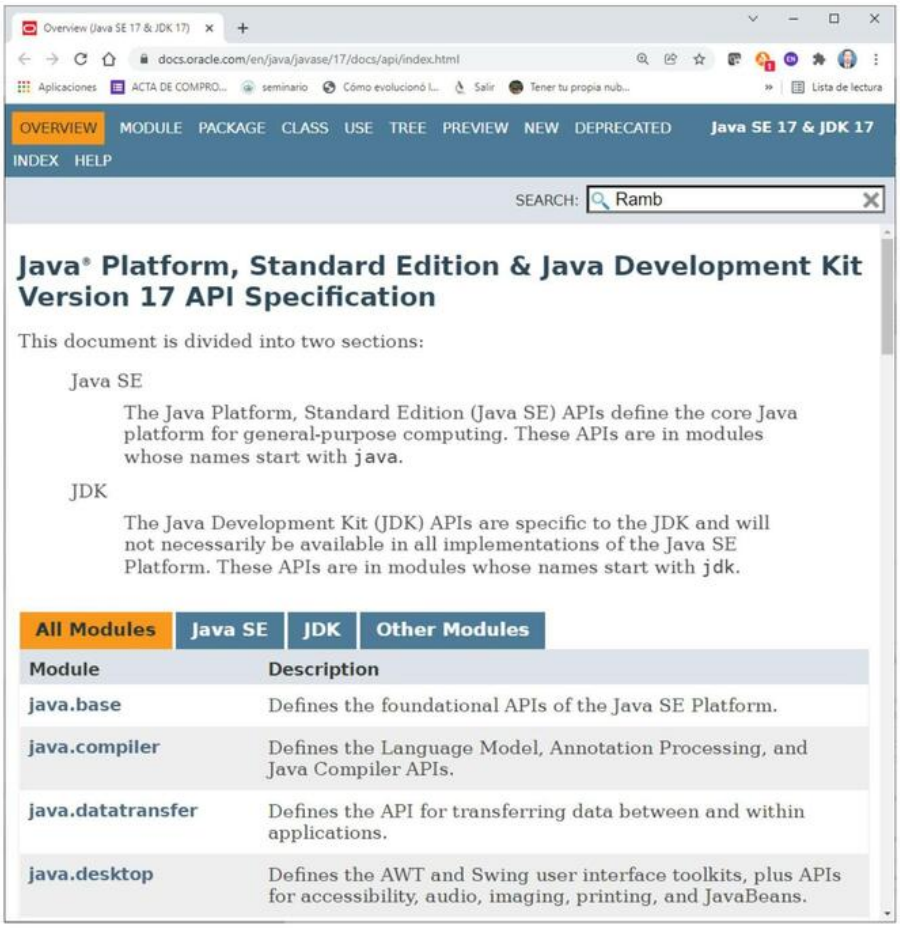
14.6 Lecturas recomendadas

- Revisar la URL https://docs.oracle.com/en/java/javase/17/docs/api/index.html.
- Convenciones de código para el lenguaje de programación.

14.7 Preguntas de revisión de conceptos

- ¿Por qué documentar una aplicación en Java?
- ¿Cuál es la importancia de la documentación de un proyecto de Java?

Figura 33.
API Java 8



14.8 Ejercicios

1. Realice una descripción breve de cada una de las siguientes etiquetas, dentro de las más utilizadas para documentar una aplicación de software con Javadoc.

Etiqueta	Descripción
@author	
@deprecated	
@param	
@return	
@see	
@since	
@throws	
@Version	

2. Utilizando Javadoc, documente la aplicación de software del capítulo anterior, denominada Centro Médico El Buen Samaritano.

14.9 Bibliografía

https://docs.oracle.com/en/java/javase/17/docs/api/index.ht

PROGRAMACIÓN ORIENTADA A OBJETOS EN JAVA

En el quehacer diario de la docencia, y más concretamente en el área de la programación, surgen muchos temas, talleres y ejercicios para trabajar con los estudiantes. En tal sentido, este libro es el resultado de varios años de experiencias donde se retroalimentan carencias y éxitos en busca de buenas prácticas de la Programación Orientada a Objetos (POO); para tal efecto, se utiliza Java como lenguaje de programación de alto nivel. Está diseñado para cualquier estudiante que desea hacer inmersión en la programación de computadores, independientemente del área o disciplina del conocimiento, pero, eso sí, cambiando el paradigma de la programación estructurada a la de los objetos, como en la vida real.

Se tienen en cuenta elementos de buenas prácticas de la programación con la finalidad de hacer fácil la lectura del código, así como su mantenimiento, facilitando la escalabilidad del mismo, la reutilización y la integración de manera homogénea, estandarizando el desarrollo, basado en convenios y reglas sencillas, y aportando higiene en la codificación de los programas fuente.

- Guía práctica
- Ejemplos desarrollados
- Ejercicios propuestos

Miguel Hernández Bejarano

Ingeniero de sistemas de la Universidad Autónoma de Colombia, con especialización en diseño y soluciones telemáticas de la misma Universidad, Magíster en comercio electrónico del Instituto Tecnológico de Monterrey (México) Magister en seguridad informática y Magíster en Ingeniería de Software de la Universidad Internacional de La Rioja (España), cursando estudios de Doctorado en Ingeniería de Sistemas e Informática. Profesor investigador y coinvestigador de varios proyectos de investigación; con reconocimiento de la Sociedad Latinoamericana de Ciencia y Tecnología en el 2015-2019. ORCID https://orcid.org/0000-0001-8509-6731

Luis Eduardo Baquero Rey

Graduado como Ingeniero de Sistemas de la Universidad Autónoma de Colombia, con maestría en Auditoría de Sistemas y Computación de la Universidad Santo Tomás (Colombia) y en Seguridad Informática de la Universidad Internacional de La Rioja (España). Actualmente cursa su segundo año del doctorado en Ingeniería de Sistemas e Informática en la Universidad de Zaragoza (España), Profesor investigador y Par evaluador reconocido por Minciencias. ORCID iD: https://orcid.org/0000-0002-2520-1541

Ediciones de la U

ISBN: 978-958-792-463-3

9 789587 924633