

Introducción a la programación orientada a objetos

Jorge Cervantes O., María del Carmen Gómez F.,
Pedro Pablo González P. y Abel García N.

-
- Clases y objetos
 - Encapsulamiento
 - Herencia
 - Polimorfismo

Obra ganadora del Segundo Concurso para la publicación de libros de texto
y materiales de apoyo a la impartición de los programas de estudio
de las licenciaturas que ofrece la Unidad Cuajimalpa

Introducción a la programación orientada a objetos

2) Introducción a la programación orientada a objetos

Clasificación Dewey: 005.117 I58

Clasificación LC: QA76.64 I58

Introducción a la programación orientada a objetos / Jorge Cervantes Ojeda ... [et al.] ; corrección de estilo y cuidado editorial Hugo A. Espinoza Rubio. -- México : UAM, Unidad Cuajimalpa, c2016.

198 p. : il. col., diagrs. ; cm.

ISBN: 978-607-28-0829-4

1. Programación orientada a objetos (Computadoras). 2. Java (Lenguaje de programación para computadora). 3. Estructura de datos (Computadora).

I. Cervantes Ojeda, Jorge, coaut. II. Espinoza Rubio, Hugo A., colab.

Esta obra fue dictaminada positivamente por pares académicos mediante el sistema doble ciego y evaluada para su publicación por el Consejo Editorial de la UAM Unidad Cuajimalpa.

© 2016 Por esta edición, Universidad Autónoma Metropolitana, Unidad Cuajimalpa
Avenida Vasco de Quiroga 4871

Col. Santa Fe Cuajimalpa, delegación Cuajimalpa de Morelos

C.P. 05348, Ciudad de México (Tel.: 5814 6500)

www.cua.uam.mx

ISBN: 978-607-28-0829-4

Primera edición: 2016

Corrección de estilo: Hugo A. Espinoza Rubio

Diseño editorial y portada: Ricardo López Gómez

Ninguna parte de esta obra puede ser reproducida o transmitida mediante ningún sistema o método electrónico o mecánico sin el consentimiento por escrito de los titulares de los derechos.

Impreso y hecho en México

Printed and made in Mexico

JORGE CERVANTES OJEDA, MARÍA DEL CARMEN GÓMEZ FUENTES,
PEDRO PABLO GONZÁLEZ PÉREZ Y ABEL GARCÍA NÁJERA

Introducción a la programación orientada a objetos



UNIVERSIDAD AUTÓNOMA METROPOLITANA

Dr. Salvador Vega y León

Rector General

M. en C. Q. Norberto Manjarrez Álvarez

Secretario General

Dr. Eduardo Peñalosa Castro

Rector de la Unidad Cuajimalpa

Dra. Caridad García Hernández

Secretaria de la Unidad Cuajimalpa

Dra. Esperanza García López

Directora de la División de Ciencias de la Comunicación y Diseño

Dr. Raúl Roydeen García Aguilar

Secretario Académico de la División de Ciencias de la Comunicación y Diseño

Dr. Hiram Isaac Beltrán Conde

Director de la División de Ciencias Naturales e Ingeniería

Dr. Pedro Pablo González Pérez

Secretario Académico de la División de Ciencias Naturales e Ingeniería

Dr. Rodolfo Suárez Molnar

Director de la División de Ciencias Sociales y Humanidades

Dr. Álvaro Peláez Cedrés

Secretario Académico de la División de Ciencias Sociales y Humanidades

Índice

Introducción.....	9
El programa de estudio de Programación Orientada a Objetos.....	10
Conocimientos, habilidades y actitudes a desarrollar	11
Objetivos.....	11
El paradigma de la Programación Orientada a Objetos	13
Objetivo.....	13
¿Qué significa paradigma?.....	13
Las ventajas de la POO	14
Los objetos.....	15
Las clases.....	16
Ejemplos de objetos.....	19
Cuestionario	21
Introducción a Java	23
Objetivo.....	23
El lenguaje Java	23
La arquitectura de ejecución de programas escritos en Java.....	23
Java Development Kit (JDK)	25
Clases en Java	25
Los atributos	26
Los métodos	27
Crear un objeto	28
Ejercicios resueltos	31
La palabra reservada <code>this</code> y los métodos "getters" y "setters"	35
Los métodos constructores.....	37
Objetos dentro de los objetos.....	42
Wrappers	47
Definición de constantes.....	48
Cadenas de caracteres	49

Ejercicios resueltos	51
Prácticas de laboratorio	59
Cuestionario	62
Relaciones entre clases	63
Objetivo.....	63
Introducción.....	63
La generalización (herencia)	64
Generalización y especialización.....	66
Existencia de la relación de herencia	67
Implementando la herencia.....	68
Las clases abstractas	71
Ejercicios de herencia.....	73
Niveles de asociación	81
La asociación simple	81
La agregación	82
La composición.....	84
Más ejemplos de composición	85
Ejercicios resueltos	86
Los métodos de <code>RelojDeManecillas</code> son:	90
Los métodos de <code>Cucu</code> son:.....	94
Prácticas de laboratorio	105
Cuestionario	109
Uso de algunas clases predefinidas en Java.....	111
Objetivos.....	111
Introducción.....	111
Excepciones	111
Entrada y salida de datos como texto (teclado y pantalla)	114
Lectura de datos del teclado mediante <code>InputStreamReader</code>	114
Lectura de datos del teclado mediante <code>Scanner</code>	118
Persistencia de objetos mediante serialización	121
Ejercicios de entrada/salida de datos	124
Arreglos en Java (listas homogéneas)	130
Arreglos como parámetros de entrada a un método	133
Arreglos como valor de retorno de un método	134
Ejercicios con arreglos	135
Listas con objetos.....	145
Ejercicios con listas ligadas.....	147

Clases para pilas y colas	154
Ejercicios de pilas.....	155
Concepto de fila (cola)	159
Ejemplos de filas (colas).....	160
Prácticas de laboratorio	162
Cuestionario	164
Polimorfismo y herencia múltiple	165
Objetivos.....	165
Polimorfismo.....	165
Interfaces.....	170
Propiedades de una interfaz	172
Interfaces múltiples	172
Polimorfismo por interfaces.....	174
Herencia múltiple	175
La herencia múltiple por interfaces.....	175
Prácticas de laboratorio	186
Cuestionario	187
Apéndice 1. Código	189
Clase OneWay	189
Clase Interfono	190
Clase WalkieTalkie	190
Clase TwoWay	191
Clase HomePhone	193
Clase Celular	194
Fuentes	195
Glosario.....	196

Introducción

La Programación Orientada a Objetos (POO) es útil cuando un sistema se modela de forma casi análoga a la realidad, porque con ésta se simplifica el diseño de alto nivel. La POO es una de las técnicas de programación más utilizadas en la actualidad, por lo que su estudio es fundamental.

En este libro exponemos los principios del paradigma orientado a objetos y presentamos problemas de diseño y construcción de programas bajo este paradigma mediante el uso del lenguaje de POO Java. Tomamos en consideración las características del modelo educativo de la Universidad Autónoma Metropolitana Cuajimalpa (UAM-C), incorporando una metodología a la que llamamos solución por etapas (Gómez, Cervantes y García, 2012: 218-223), similar a la de Niklaus Wirth (1971: 221-227), la cual fomenta la habilidad de autoaprendizaje en la programación. Esta metodología consiste en mostrar una o varias etapas de la solución de un ejercicio, hasta llegar a la solución completa.

Cuando la solución se brinda por etapas, los ejercicios son realmente ejemplos en los que el alumno estudia las soluciones parciales planteadas, lo cual le hace pensar en cómo dar con la solución completa, pero ya con una parte resuelta que le indica, poco a poco, una manera bien estructurada de resolver el ejercicio.

Cada capítulo contiene sus objetivos particulares y prácticas como experiencias de aprendizaje, las cuales están especialmente diseñadas para que el alumno refuerce los conocimientos adquiridos. Con la intención de facilitar la comprensión de los temas, todas las explicaciones están acompañadas de figuras y ejemplos. También incluimos dentro del código recuadros con aclaraciones que proporcionan más detalles en los lugares clave. Los ejercicios y ejemplos de este libro son el resultado de la mejora continua a lo largo de ocho años de experiencia impartiendo la UEA de Programación Orientada a Objetos en la UAM-C. Además, cada capítulo contiene un cuestionario que facilita la autoevaluación y la síntesis de los conocimientos que expone.

Nuestro objetivo es que este libro sea de ayuda no sólo para alumnos y profesores de la UAM-C, sino para todos los que deseen aprender los principios de la POO.

El programa de estudio de Programación Orientada a Objetos

Para desarrollar este libro, nos basamos en el programa de estudio de la UEA Programación Orientada a Objetos de la Licenciatura de Ingeniería en Computación, el cual se detalla a continuación:

Contenido sintético

1. Introducción.

- 1.1 Origen del paradigma orientado a objetos.
- 1.2 Definición de objeto y clase.
- 1.3 Elementos de una clase: atributos y métodos.
- 1.4 Instanciación, uso y destrucción de objetos.
- 1.5 Arreglos de objetos.
- 1.6 Métodos Constructores y Destructores. Recolección de basura.
- 1.7 Encapsulación (visibilidad).

2. Uso de clases predefinidas básicas.

- 2.1 Clases para cadenas de caracteres.
- 2.2 Clases para entrada y salida de datos (teclado, pantalla).
- 2.3 Clases para funciones matemáticas.

3. Herencia y Polimorfismo.

- 3.1 Generalización vs. Especialización.
- 3.2 Jerarquías de clases y herencia (Superclase y subclase).
- 3.3 Clases abstractas y concretas.
- 3.4 Redefinición de métodos (sobreescritura).
- 3.5 Herencia múltiple.
- 3.6 Polimorfismo.

4. Uso de clases predefinidas avanzadas.

- 4.1 Clases para listas homogéneas y heterogéneas, conjuntos.
- 4.2 Clases para entrada y salida de datos (disco).
- 4.3 Clases para pilas, colas.

5. Manejo de excepciones

- 5.1 Recepción de excepciones.
- 5.2 Lanzamiento de excepciones.

El programa de estudio contempla la posibilidad de que la UEA se imparta en cualquier lenguaje de programación (Java, C++, C#, etc.). Este libro de texto se especializa en la enseñanza de la POO con Java y cubre al cien por ciento el plan de estudio de dicha UEA. Sin embargo, cabe señalar que algunos de los conceptos incluidos en el programa (destructores, recolección de basura) no aplican para Java.

Conocimientos, habilidades y actitudes a desarrollar

La POO no sólo es muy útil para elaborar sistemas de cómputo complejos, sino que también es la base para el desarrollo de aplicaciones web. Los empleos para los programadores web están muy bien remunerados hoy en día, debido al predominio que tiene la Internet. Entonces, podemos decir que el estudio de la POO es de gran interés para todos los que cursan carreras relacionadas con el mundo de las computadoras.

Un buen programador debe tener la capacidad de abstraer problemas y solucionarlos mediante la computadora. Los ejemplos y ejercicios del presente libro fomentan el desarrollo de esta habilidad. Los casos de estudio que se incluyen favorecen el desarrollo de las habilidades analíticas y de comprensión que el estudiante requiere para trabajar con el paradigma orientado a objetos.

Una de las habilidades de un egresado de la Licenciatura de Ingeniería en Computación, según el perfil de egreso, es aplicar modelos y técnicas para diseñar, implementar y probar sistemas de software de forma eficiente. Este libro ayuda en gran medida al desarrollo de esta habilidad, particularmente con el paradigma orientado a objetos.

Objetivos

- General: desarrollar en el alumno la habilidad de construir programas bajo el paradigma orientado a objetos.
- Específicos: comprender los principios del paradigma orientado a objetos, así como resolver problemas de construcción de programas bajo el paradigma orientado a objetos.

Para comprender mejor este libro, como conocimientos previos, se recomienda que el lector haya cursado programación estructurada.

El paradigma de la Programación Orientada a Objetos

El desarrollo orientado a objetos es una tecnología ya probada y se ha utilizado para construir y entregar multitud de sistemas complejos en una amplia gama de dominios de problemas. La demanda de software complejo crece vertiginosamente. El valor fundamental del desarrollo orientado a objetos es que libera a la mente humana para que pueda centrar sus energías creativas sobre las partes realmente difíciles de un sistema complejo.

GRADY BOOCHE (1998)

Objetivo

Ubicar el paradigma de la POO como una forma eficaz de elaborar sistemas de cómputo.

¿Qué significa paradigma?

Un *paradigma* es una metodología que intenta unificar y simplificar la manera en que se resuelve un cierto grupo de problemas. En el contexto de la programación, un paradigma es un conjunto de principios y métodos que sirven para resolver los problemas a los que se enfrentan los desarrolladores de software al construir sistemas grandes y complejos. Existen varios paradigmas de programación; los importantes son los siguientes:

- *El estructurado.* Se basa en estructuras de control de flujo de programa (por ejemplo, si/entonces/si no, para y mientras). No se hacen "saltos" de un lugar a otro dentro de una rutina. De esta manera, los programas son más fáciles de entender.
- *El funcional.* Se programa con funciones y sus llamados correspondientes. El código con funciones pequeñas queda muy claro y promueve la reutilización.
- *El orientado a objetos.* Los programas trabajan con base en unidades llamadas objetos, los cuales siguen una serie de principios que veremos más adelante.

Las ventajas de la POO

El paradigma orientado a objetos es útil cuando el sistema se modela de forma casi análoga a la realidad, porque así se simplifica el diseño de alto nivel. Esta analogía permite que los programadores tengan más claro cuál es el papel de cada porción del programa y de los datos, lo que facilita la creación y el mantenimiento del sistema. Además, se promueve la reutilización, pues las similitudes entre objetos se programan sólo una vez en forma abstracta y el programador concentra su esfuerzo en las diferencias concretas.

En la POO podemos, por ejemplo, diseñar el código para un botón virtual genérico que detecta el "click" del mouse y llama a una función. El código del botón se reutiliza cada vez que queremos crear un botón, pero con características particulares para cada caso, como se observa en la figura I-1.



Figura I-1. El código del botón virtual genérico se reutiliza para cada botón particular

En general, el mantenimiento se facilita con el paradigma orientado a objetos, porque el software queda bien organizado y protegido. De esta manera, un programador entiende mejor el código de otro y hay menor riesgo de que sus cambios afecten el trabajo de los demás.

También se mejora el desarrollo de software a gran escala. Los equipos de programadores trabajan sobre objetos diferentes y, posteriormente, se integra el trabajo de todos haciendo uso de las interfaces (la cara hacia afuera) de los objetos.

En la figura I-2 se comparan las características principales de la programación estructurada y de la POO. Cabe señalar que no es mejor una que la otra, sino que su utilidad depende del tipo de sistema que se desarrollará. Si queremos utilizar la POO en todos los casos, complicaríamos el problema en lugar de facilitar su solución.

Así que debe procurarse que no suceda lo que dice el refrán: “El que tiene un martillo, suele verle a todo cara de clavo”.

<i>Programación estructurada</i>	<i>Programación orientada a objetos</i>
Problemas de naturaleza algorítmica (dada cierta entrada se produce una cierta salida)	Desarrollo de aplicaciones web y otros sistemas que se presten al modelado de objetos.
Se basa en estructura de datos	Se basa en objetos, que tienen un estado y un comportamiento.
La información fluye a través de las estructuras y de llamados a funciones.	Hay interacciones entre los objetos

Figura I-2. Programación estructurada vs. programación orientada a objetos

Los objetos

En la POO, un objeto es una entidad virtual (o entidad de software), con datos y funciones que simulan las propiedades del objeto. Los objetos con los que se construyen los programas se ven como si fueran máquinas, las cuales están formadas por un conjunto de elementos autónomos. Las propiedades individuales de estos elementos y las relaciones entre sí definen el funcionamiento general de la máquina.

Desde el punto de vista del mundo real, un objeto tiene dos propiedades esenciales: un estado y un comportamiento:

- *El estado.* Son los datos asociados con el objeto, los cuales indican su situación interna en un momento dado, por ejemplo: velocidad, calificación, color, capacidad, encendido/apagado, saldo, etc.
- *El comportamiento.* Es la manera en la que el objeto responde a estímulos del exterior, por ejemplo, lo que sucede cuando se oprime el botón “inicio”, lo que sucede cuando se hace un retiro de una cuenta bancaria o cuando se oprime el botón “reiniciar” en un contador.

Desde el punto de vista computacional:

- Los atributos. Son los datos que pertenecen al objeto y que representan el estado de éste, en función de los valores que tienen, por ejemplo:

```
double saldo;
float calificacion;
boolean on;
```

- *Los métodos:* Definen el comportamiento del objeto y son funciones que se pueden invocar desde otros objetos. Los métodos pueden modificar el estado del objeto cuando cambian el valor de alguno de los atributos, por ejemplo:

```
double obtenerSaldo();
float calcularPromedio();
boolean seOprimioOnOff();
```

Por ejemplo, en la figura I-3 se muestra el objeto *cuenta bancaria*, cuyos datos contenidos son el nombre del cuentahabiente, sus apellidos, dirección, mail, saldo y el *tipo de cuenta*, en donde se especifica si es una cuenta de crédito, débito, ahorro, etc. Además, el objeto *cuenta bancaria* también contiene algunas operaciones que se pueden ejecutar con la cuenta, en este ejemplo son consultar saldo, retiro, bonificación, actualizar datos y consultar datos.

El objeto cuenta bancaria tiene:

<i>Los datos del cuentahabiente</i>	<i>las operaciones que se pueden hacer con la cuenta</i>
Nombre	Consultar saldo
Apellidos	Retiro
Dirección	Bonificación
Mail	Actualizar datos
Saldo	Consultar datos
Tipo de cuenta	

Figura I-3. Ejemplo de objeto: cuenta bancaria

A los datos del objeto se les denomina *atributos* y a las operaciones que se pueden ejecutar con el objeto se les llama *métodos*. Entonces, todos los objetos tienen un *estado* (conjunto de atributos) y un *comportamiento* (conjunto de métodos).

Las clases

Cada objeto en la POO tiene propiedades definidas por su clase de objeto.

Una clase es un tipo de dato definido por el programador específicamente para crear objetos. Se dice que cada objeto es una *instancia* particular de alguna *clase* de objetos. La clase define las propiedades comunes de un conjunto de objetos. El programador define una clase como lo hace con un tipo de dato compuesto y le da un nombre. Una vez definida la clase, los objetos se crean a partir de ésta. En la figura I-4 se ilustra que se pueden crear varios reloj a partir de una misma clase.

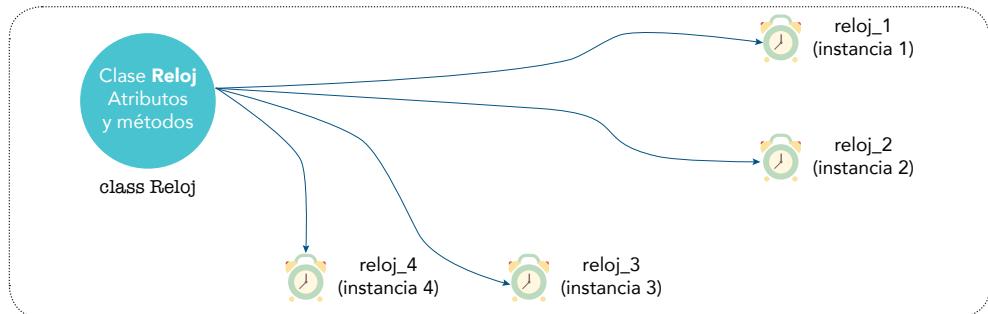


Figura I-4. Varias instancias de una misma clase

En programación, a las instancias de una clase se les llama *objetos*. A continuación, como ejemplo, definimos la clase `Reloj`. Por convención, los nombres de las clases comienzan siempre con mayúscula. Un reloj se caracteriza por tener tres indicadores: la hora, los minutos y los segundos; por lo tanto, los atributos de la clase `Reloj` son tres números enteros. El comportamiento de este reloj está definido por las siguientes operaciones: inicializar la hora con un valor dado, incrementar la lectura actual en un segundo y obtener la hora. Estas operaciones se definen mediante los métodos `set`, `incrementar` y `getHora`.

```
public class Reloj {
    private int hora;
    private int minuto;
    private int segundo;

    public void set( int h, int m, int s ) {
        hora     = h % 24;
        minuto   = m % 60;
        segundo = s % 60;
    }

    public void incrementar() {
        segundo = ( segundo + 1 ) % 60;
        if ( segundo == 0 ) {
            minuto = ( minuto + 1 ) & 60;
            if ( minuto == 0 ) {
                hora = ( hora + 1 ) % 24;
            }
        }
    }

    public String getHora() {
        return hora + ":" + minuto + ":" + segundo;
    }
}
```

En el segundo capítulo, “Introducción a Java”, estudiaremos la forma de crear objetos en dicho lenguaje. Como se observa en la figura I-5, al crear los objetos de clase `Reloj`, lo que importa hacia el exterior es la *interfaz* del objeto, es decir, la manera en la que éste es operado desde el exterior. Por ello los atributos del ejemplo anterior están declarados como *privados* y sus métodos como *públicos*. La idea es que, como en la vida real, los objetos no sean “abiertos” y que sólo se manipulen a través de la interfaz diseñada para hacerlo. De esta forma, se aseguraría que serán operados adecuada y seguramente.

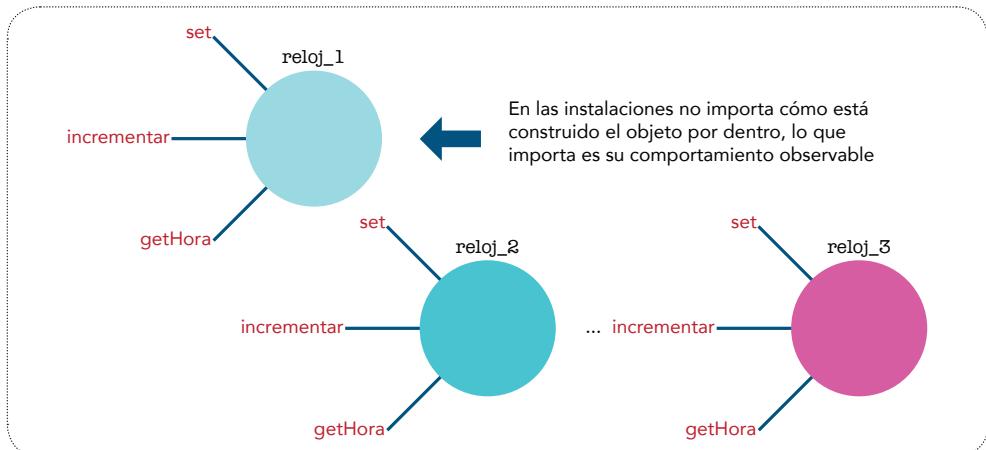


Figura I-5. La construcción interna del objeto no importa hacia el exterior

La clase es la descripción de un conjunto de objetos similares, es decir, que comparten los mismos atributos y los mismos métodos. La clase es el bloque constructor más importante de cualquier lenguaje de POO. El Unified Modeling Language, UML (Lenguaje Unificado de Modelado) (Booch *et al.*, 1998) se utiliza para el modelado orientado a objetos, el cual es tema de un curso posterior. Sin embargo, para comenzar a familiarizar al estudiante con este lenguaje, ilustraremos las clases con notación UML. En la figura I-6 se muestra la representación UML de la clase `Reloj`.

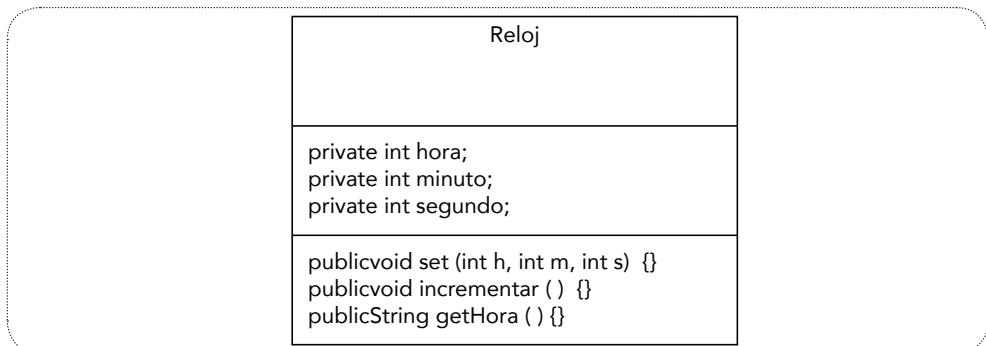


Figura I-6 Representación de la clase `Reloj` en UML

En el primer recuadro se indica el nombre de la clase; en el segundo, los atributos que contiene, mientras que en el tercero, sus métodos.

Ejemplos de objetos

A continuación daremos algunos ejemplos de cómo modelar objetos o conceptos de la vida real.

Ejemplo 1. Modelando una lámpara (luz) como objeto

En la figura I-7, **luz** es un objeto con los métodos **encender()**, con el que se enciende la luz; **apagar()**, con el que se apaga la luz; **estaPrendida()**, con el que se verifica si la luz está encendida; **setIntensidad()**, con el que se establece la intensidad, y **getIntensidad()**, con el que se obtiene el valor actual de intensidad. El estado del objeto está compuesto por su intensidad actual y si está encendido o apagado y se conoce sólo a través de los métodos **estaPrendida()** y **getIntensidad()**.

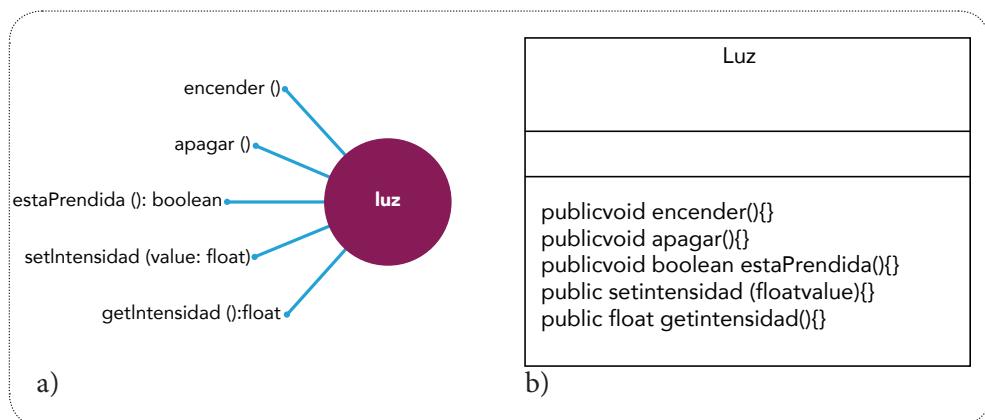


Figura I-7. a) Representación gráfica del objeto **luz**; b) representación de la clase **Luz** en UML

Ejemplo 2. Modelando un contador como objeto

En la figura I-8, **contador** es un objeto con los métodos **incrementar()**, con el que se incrementa el contador; **setValor()**, con el que se establece el valor actual del contador, y **getValor()**, con el cual se recupera el valor actual del contador. El estado del objeto es el valor actual de conteo, que se conoce solamente a través del método **getValor()**.

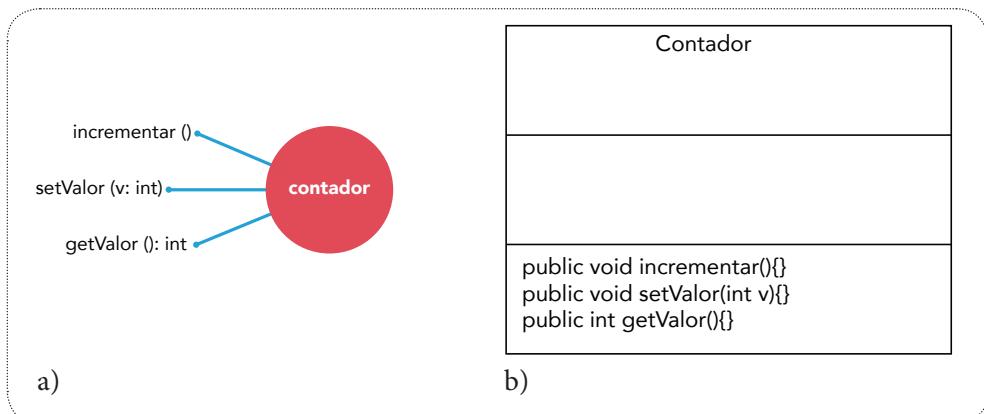


Figura I-8. a) Representación gráfica del objeto `contador`; b) representación de la clase `Contador` en UML

Ejemplo 3. Modelando una lista como objeto

En la figura I-9, `lista` es un objeto con los métodos `insertar()`, con el que se agrega un nuevo elemento en la lista; `remover()`, con el que se elimina el elemento en la posición `indice` de la lista; `getElemento()`, con el que se obtiene el elemento en la posición `indice` de la lista, y `getTamanio()`, con el que se obtiene el número de elementos de la lista.

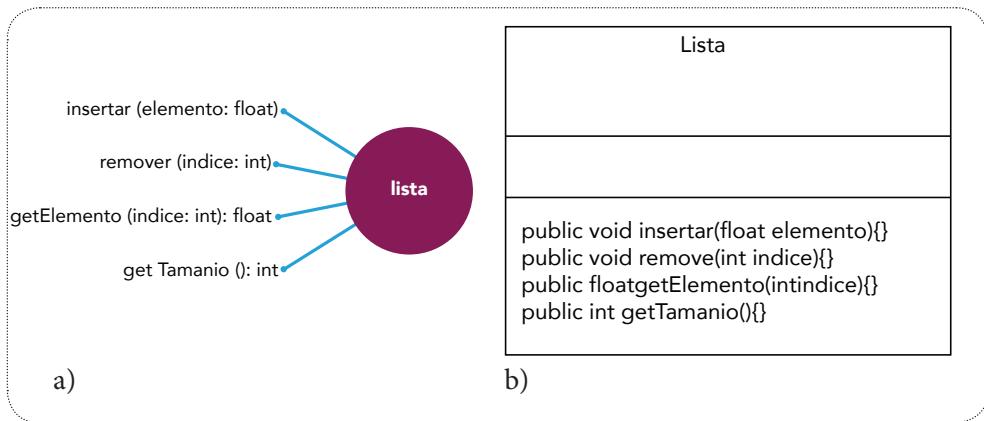


Figura I-9. a) Representación gráfica del objeto `lista`; b) representación de la clase `Lista` en UML

Ejemplo 4. Modelando una tarjeta de crédito como objeto

En la figura I-10, `tarjetaDeCredito` es un objeto con los métodos `getLimiteCredito()`, `getSaldoDeudor()`, `efectuarCargo()`, el cual es positivo para un abono y negativo para un retiro; `calcularIntereses()`, con el que se calculan los nuevos intereses generados; `calcularPagoMinimo()`, que calcula el pago mínimo que debe efectuarse entre la fecha de corte y la fecha de pago,

y generarEstadoDeCuenta(), que genera toda la información producida al momento del corte.

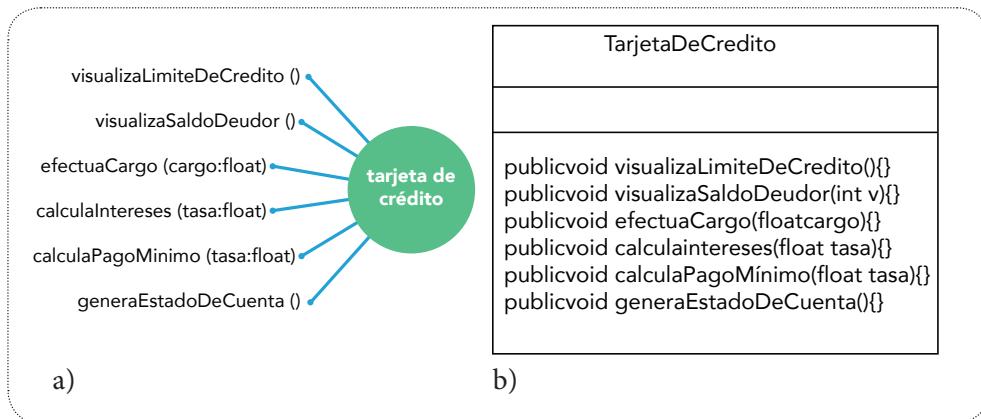


Figura I-10. a) Representación gráfica del objeto `TarjetaDeCredito`; b) representación de la clase `TarjetaDeCredito` en UML

Cuestionario

Contesta las siguientes preguntas:

1. ¿Qué es un paradigma?
2. ¿Cuándo es útil la POO?
3. Menciona tres ventajas de la POO.
4. ¿Qué es mejor, la programación estructurada o la POO?
5. ¿Qué es un objeto?
6. ¿Qué define el estado y el comportamiento de un objeto?

Introducción a Java

Objetivo

Introducir al alumno a los principios básicos del lenguaje de POO Java, de tal forma que pueda construir clases y crear objetos a partir de éstas.

El lenguaje Java

Java es un lenguaje en el que se trabaja, preferentemente, sólo con clases y objetos. Esto implica que todas las variables representan instancias de objetos de alguna clase. De todas formas, los tipos primitivos como `int`, `float` y `double` se mantienen como opciones por eficiencia, aunque tienen su equivalente en forma de clase. En Java, incluso el programa en sí es un objeto. Enseguida presentamos el ejemplo de un programa en Java que escribe en la pantalla el mensaje "Hola Mundo!".

```
// Ejemplo simple
public class Ejemplo {

    // main es el metodo principal
    public static void main( String[] args ) {
        // imprime algo en la salida estandar
        System.out.println( "Hola mundo!" );
    }
}
```

La arquitectura de ejecución de programas escritos en Java

Entre las principales diferencias que distinguen a Java de otros lenguajes como C y C++, está la arquitectura de ejecución. Un programa escrito en C++ resulta en una aplicación específica para una máquina y sistema operativo; mientras que un programa en Java corre sobre una máquina virtual llamada Java Virtual Machine (JVM), la cual es un programa que funciona como si fuera una computadora y que, por lo tanto, puede utilizarse para ejecutar programas.

Existen versiones de la JVM para cualquier máquina y sistema operativo, por lo que los programas en Java corren sin necesidad de modificarse en cualquier máquina física que ejecute la JVM. Esta particularidad ha convertido a Java en uno de los

lenguajes de programación más populares del mundo, pues existen muchos tipos y marcas de hardware, así como de sistemas operativos que cuentan con una JVM, lo cual multiplica el valor del producto de los programadores al ser sus programas de los más ejecutados.

La figura II-1 ilustra la diferencia entre la arquitectura de ejecución de un programa escrito en C/C++ y la de un programa en Java:

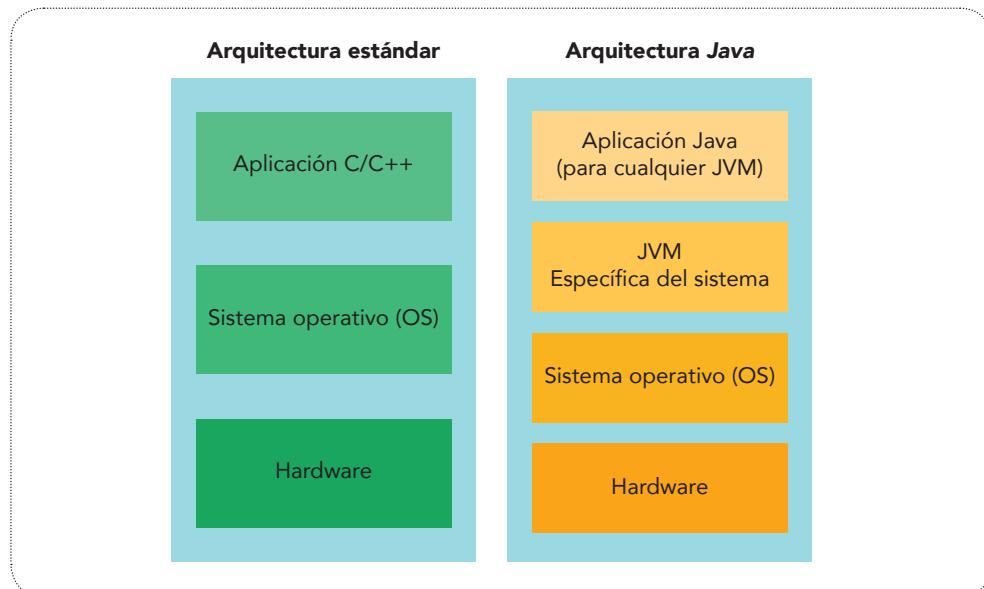


Figura II-1. Diferentes arquitecturas de ejecución

La figura II-2 muestra los productos parciales obtenidos después de cada etapa en la creación de software: tanto en una compilación estándar, como en una compilación Java. El código en Java compilado tiene la extensión `.class` y el archivo está escrito en *bytecode*. La máquina virtual de cada sistema operativo interpreta este bytecode ejecutable:

- Los programas fuente en Java tienen extensión `.java`
- Los archivos comprimidos tienen la extensión `.jar`

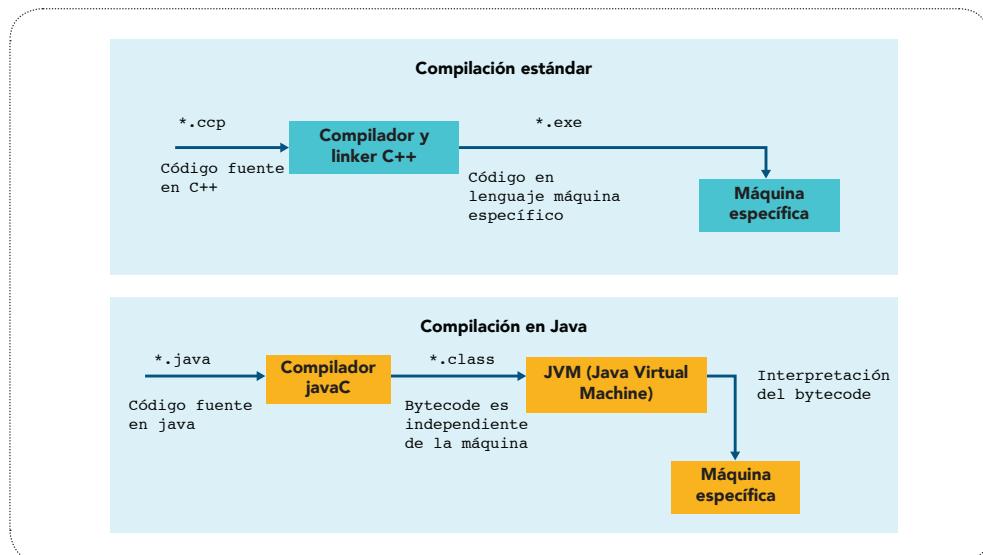


Figura II-2. Compilación estándar vs. compilación en Java

Java Development Kit (JDK)

El Java Development Kit (JDK) es el conjunto de herramientas que nos permite trabajar en Java e incluye las herramientas y librerías para desarrollar aplicaciones y documentarlas. El JDK contiene, entre otras cosas, el Java Runtime Environment (JRE) que consta de la JVM y de librerías. Las principales herramientas que contiene el JDK son

- `javac`, compilador Java
- `java`, Java Virtual Machine, para ejecutar aplicaciones Java
- `javadoc`, para crear automáticamente documentación Java en HTML
- `jar`, para compactar y descompactar archivos

Clases en Java

La sintaxis para definir una clase en Java es la siguiente:

```
public class <NombreClase> {
    <definicion de los atributos>
    <definicion de los constructores>
    <definicion de los metodos>
}
```

Por convención, el nombre de una clase debe tener inicial mayúscula y el resto en minúsculas. En el caso de nombres compuestos, se utilizan las iniciales de cada palabra en mayúscula.

Los atributos

Los atributos definen la estructura de datos de la clase, los cuales, por omisión, son *públicos*, es decir, accesibles desde otras clases, lo que significa que se modifican desde afuera del objeto. Es altamente recomendable declarar todos los atributos con el modificador `private` y solamente cambiarlo a `public` o `protected` cuando sea absolutamente necesario. En los ejemplos de este libro se verá más claramente el uso de estos modificadores.

Por convención, los nombres de los atributos deben escribirse en minúsculas. En el caso de nombres compuestos, cada palabra intermedia debe iniciar con mayúscula. Por ejemplo:

```
private int promedio;
public float saldoCuentaCredito;
```

La figura II-3 muestra la clase `TarjetaCredito`, que contiene varios ejemplos de atributos.

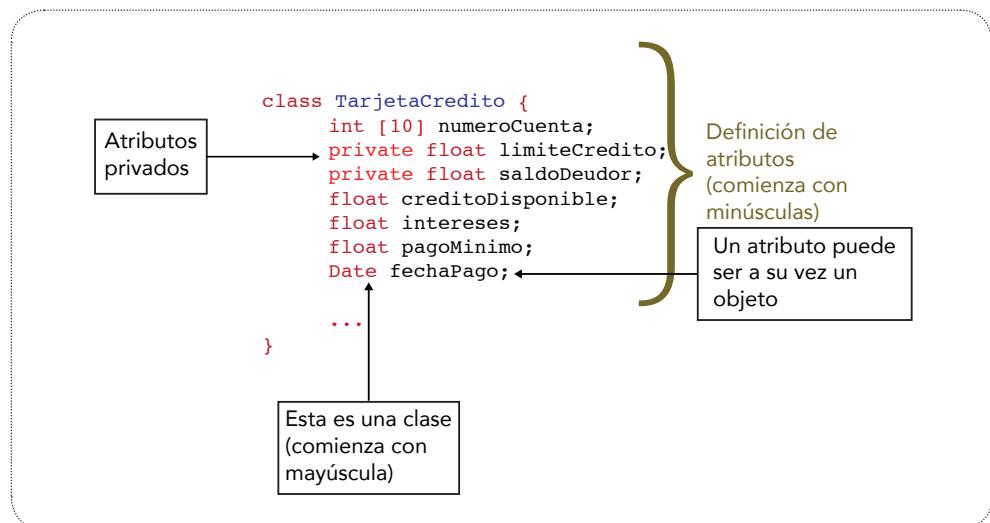


Figura II-3. Ejemplos de atributos de la clase `TarjetaCredito`

Los métodos

Los métodos constituyen el comportamiento de los objetos de la clase. La definición de un método es muy similar a la de una función. Los métodos públicos son las *operaciones* que los objetos externos realizan con el objeto en cuestión. Los métodos privados son las *operaciones internas* que no se pueden invocar desde el exterior, pero sí desde otro método dentro de la clase.

La sintaxis para escribir un método público es la siguiente:

```
public <TipoDatoRetorno> <NombreMetodo> ( <listaParametros> ) {  
    ...  
}
```

Y para escribir un método privado es similar, sólo cambia la primera palabra:

```
private <TipoDatoRetorno> <NombreMétodo> ( <listaParametros> ) {  
    ...  
}
```

Los métodos privados son auxiliares de los públicos. Se escriben para estructurar mejor el código de la clase. Por convención, los nombres de los métodos son verbos en infinitivo y escritos en minúsculas. En el caso de nombres compuestos, se usan mayúsculas para la inicial de cada palabra intermedia.

En el siguiente código se declara la clase `Ejemplo`, con los atributos `x` y `a`. Esta clase también contiene los métodos `hacerAlgo()`, `suma()` e `imprimir()`.

```
public class Ejemplo {  
    public int x;  
    public int a;  
  
    public void hacerAlgo() {  
        x = 1 + a * 3;  
    }  
  
    public int suma() {  
        return x + a;  
    }  
  
    public void imprimir() {  
        System.out.println( "x= " + x + " a= " + a + "\n" );  
    }  
}
```

Crear un objeto

Declaremos que los objetos e y f son de la clase Ejemplo:

```
Ejemplo e;
Ejemplo f;
```

Declarar las variables e y f de la clase Ejemplo no implica que se crearon los objetos. En realidad, hasta aquí, sólo se crean apuntadores a objetos de esta clase. Para crear un objeto, se usa el operador new. El operador new crea un nuevo objeto de la clase especificada (alojando suficiente memoria para almacenar los datos del objeto) y regresa una *referencia* al objeto que se creó. Esta *referencia* es, en realidad, un *apuntador oculto*. En Java, el manejo de apuntadores y el desalojo de memoria se hacen automáticamente para evitar olvidos de los programadores y, por ende, el uso explícito de apuntadores se omite. El código para crear las instancias de los objetos de la clase Ejemplo sería:

```
Ejemplo e;
Ejemplo f;
e = new Ejemplo();
f = new Ejemplo();
```

En general, la manera de crear un objeto de una clase X es la siguiente:

```
// primero declarar el objeto indicando su clase
<Clase> <nombreObjeto>;
// después crear el objeto
<nombreObjeto> = new <Clase>();
```

Lo anterior se crea en un solo enunciado:

```
<Clase> <nombreObjeto> = new <Clase>();
```

La figura II-4 ilustra lo que es una *referencia* al objeto. Con new se deposita en las variables e y f un apuntador a la clase Ejemplo. Se dice que las variables e y f son una *referencia* al objeto o, dicho de otra manera, a la dirección de memoria que le fue asignada por el sistema operativo de la memoria *heap* (o pedacería de memoria).

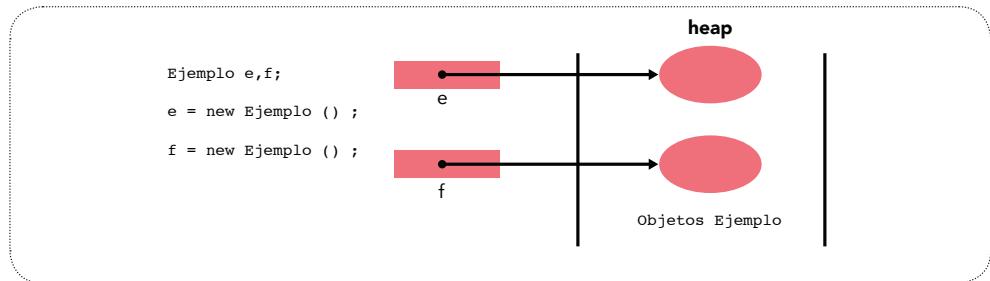


Figura II-4. Un objeto se crea con el operador new

En la práctica, sólo se dice que `e` y `f` son objetos de la clase `Ejemplo`. También podemos crear los objetos `e` y `f` en un solo paso:

```
Ejemplo e = new Ejemplo();
Ejemplo f = new Ejemplo();
```

Para tener acceso a los atributos y a los métodos del objeto, se utiliza el operador punto “`.`”. La sintaxis de la notación punto de Java para acceder a un método es la siguiente:

```
<referenciaObjeto>.<nombreMetodo>( <listaParametros> )
```

Entonces, la forma de ejecutar un método del objeto es:

```
nombreObjeto.nombreMetodo();
```

Por ejemplo, para llamar al método `hacerAlgo()` del objeto `e` se escribe:

```
e.hacerAlgo();
```

A la clase que tiene el método `main` se le llama *clase principal*. En el siguiente ejemplo tenemos la clase `PruebaEjemplo`, cuyo método `main()` hace uso de dos objetos de la clase `Ejemplo`,

```
public class PruebaEjemplo {
    public static void main( String[] args ) {
        Ejemplo e;
        Ejemplo f;
        e = new Ejemplo(); // instancia "e" de la clase Ejemplo
        f = new Ejemplo(); // instancia "f" de la clase Ejemplo
        e.a = 7;
        e.hacerAlgo();
        f.x = e.suma();
        f.a = f.x + e.a;
```

```

        e.a = f.suma();
        e.imprimir();
        f.imprimir();
    }
}

```

El método `main()` debe ser siempre estático. La palabra `static` en la definición de un método implica que no es necesario crear el objeto para llamar al método. Así, la JVM puede invocarlo. Si queremos evitar que algunos datos de la clase se modifiquen desde afuera de ésta, conviene declarar a los atributos en cuestión como *privados*. La sintaxis para declarar un atributo como privado, es la siguiente:

```

private <tipo> <nombreAtributo>;

public class Ejemplo {
    int x;
    private int a; // el atributo "a" es privado

    public void hacerAlgo() {
        x = 1 + a * 3;
    }

    public int suma() {
        return x + a;
    }

    public void imprimir() {
        System.out.println( "x= " + x + " a= " + a + "\n" );
    }
}

```

Entonces, no será posible tener acceso al atributo `a` desde el exterior de la clase y sólo los métodos de la clase tienen acceso a este atributo. Así es que, cuando los objetos `e` y `f` en la clase `PruebaEjemplo` tratan de leer o modificar el atributo `a`, ocurre un error de compilación.

```

public class PruebaEjemplo {
    public static void main( String[] args ) {
        Ejemplo e;
        Ejemplo f;
        e = new Ejemplo();
        f = new Ejemplo();
        e.a = 7;
        e.hacerAlgo();
        f.x = e.suma();
        f.a = f.x + e.a;
        e.a = f.suma();
    }
}

```

¡Error! El acceso al atributo `a` no está permitido.

```

    e.imprimir();
    f.imprimir();
}
}

```

Ejercicios resueltos

Ejercicio 2.1. ¿Qué es lo que imprimen los métodos `e.imprimir()` y `f.imprimir()` invocados desde la clase `PruebaEjemplo`?

Solución

En la figura II-5 se muestra paso a paso lo que sucede con cada uno de los atributos de los objetos `e` y `f`.

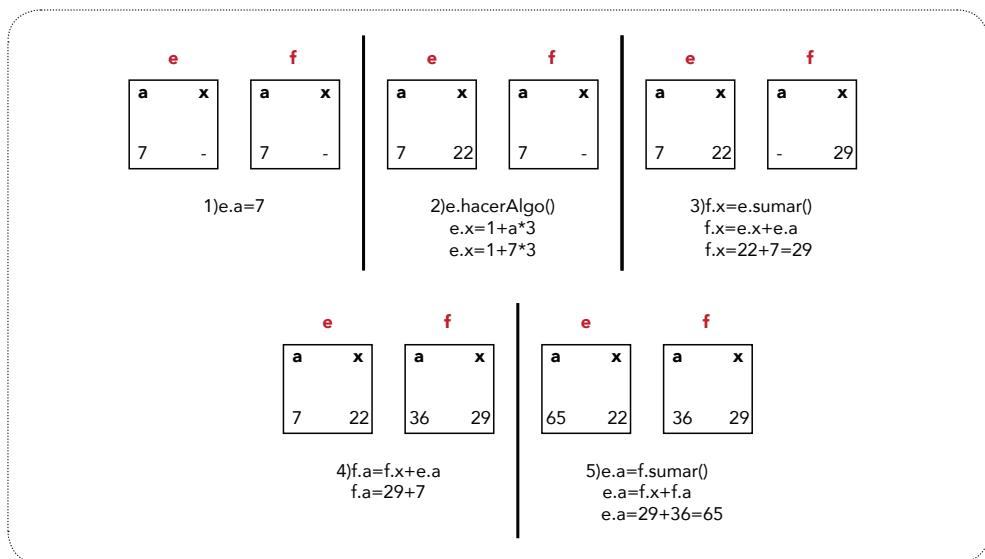


Figura II-5. Contenidos parciales de los atributos en los objetos `e` y `f`

El resultado de `e.imprimir()` y `f.imprimir()` es el contenido final de los objetos `e` y `f` de la figura II-5, es decir, para `e`: `a = 65, x = 22`; y para `f`: `a = 36, x = 29`.

Ejercicio 2.2. Declarar la clase `Cuenta` con los atributos `nombre`, `saldo`, `numero` y `tipo`, y con los métodos `depositar`, que recibe como parámetro la cantidad a depositar, y `retirar`, que recibe como parámetro la cantidad a retirar. Sólo se efectuará el retiro si el saldo es mayor o igual a la cantidad a retirar. Escribir un método que imprima los datos del objeto.

La figura II-6 es la representación de la clase Cuenta.

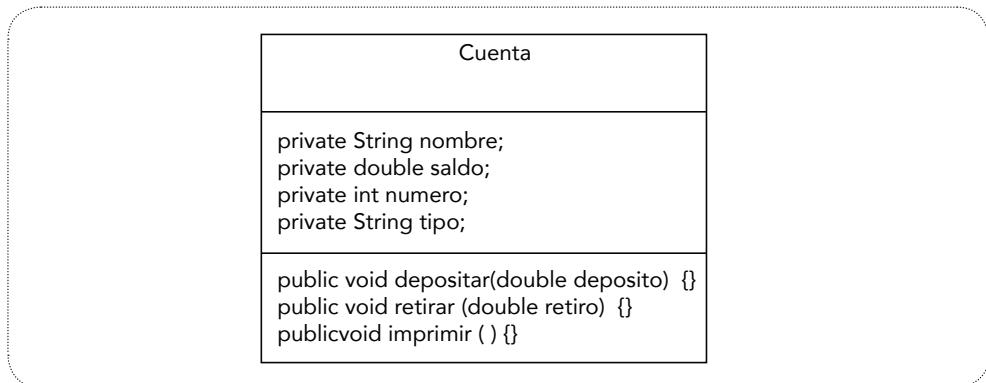


Figura II-6. Representación UML de la clase Cuenta

Solución:

```
public class Cuenta {  
    public String nombre;  
    public double saldo;  
    public int numero;  
    public String tipo;  
  
    public void depositar( double deposito ) {  
        saldo = saldo + deposito;  
    }  
  
    public void retirar( double retiro ) {  
        if ( saldo >= retiro ) {  
            saldo = saldo - retiro;  
        }  
    }  
  
    public void imprimir() {  
        System.out.println( "La cuenta es de: " + nombre  
                           + ", número: " + numero  
                           + ". Es una cuenta de " + tipo  
                           + ", con saldo: " + saldo + "\n" );  
    }  
}
```

Ejercicio 2.3. Crear los objetos `cuentaCredito` y `cuentaDebito` en la clase Principal. Al objeto `cuentaCredito` ponerlo a nombre de Pedro Sánchez, con saldo de 1500, con número de cuenta 244513, indicando que es una cuenta de crédito. Al objeto `cuentaDebito` ponerlo a nombre de Pablo García, con saldo de 7800,

con número de cuenta 273516, indicando que es una cuenta de débito. ¿Qué es lo que se imprime?

```
public class Principal {  
    public static void main( String[] args ) {  
        Cuenta cuentaCredito;  
        Cuenta cuentaDebito;  
  
        // Creamos los objetos  
        cuentaCredito = new Cuenta();  
        cuentaDebito = new Cuenta();  
  
        ...  
  
        cuentaCredito.imprimir();  
        cuentaDebito.imprimir();  
    }  
}
```

Solución:

```
public class Principal {  
    public static void main( String[] args ) {  
        Cuenta cuentaCredito;  
        Cuenta cuentaDebito;  
  
        // Creamos los objetos  
        cuentaCredito = new Cuenta();  
        cuentaDebito = new Cuenta();  
  
        cuentaCredito.nombre = "Pedro Sanchez";  
        cuentaCredito.saldo = 1500;  
        cuentaCredito.numero = 244513;  
        cuentaCredito.tipo = "crédito";  
  
        cuentaDebito.nombre = "Pablo Garcia";  
        cuentaDebito.saldo = 7800;  
        cuentaDebito.numero = 273516;  
        cuentaDebito.tipo = "débito";  
  
        cuentaCredito.imprimir();  
        cuentaDebito.imprimir();  
    }  
}
```

Ejercicio 2.3.a. Determina, sin ver la solución, cuál será la salida del programa anterior.

Solución:

```
La cuenta es de: Pedro Sanchez, número: 244513. Es una cuenta de crédito, con saldo: 1500.  
La cuenta es de: Pablo Garcia, número: 273516. Es una cuenta de débito, con saldo: 7800.
```

Ejercicio 2.4. Declarar los atributos `saldo` y `numero` de la clase `Cuenta`, como privados. Declarar también los métodos necesarios para asignarles un valor y para obtener el valor de cada cual.

Solución:

```
public class Cuenta {  
    public String nombre;  
    private double saldo;  
    private int numero;  
    public String tipo;  
  
    public void setSaldo( double s ) {  
        saldo = s;  
    }  
  
    public double getSaldo() {  
        return saldo;  
    }  
  
    public void setNumero( int num ) {  
        numero = num;  
    }  
  
    public int getNumero() {  
        return numero;  
    }  
  
    public void depositar( double deposito ) {  
        saldo = saldo + deposito;  
    }  
  
    public void retirar( double retiro ) {  
        if ( saldo >= retiro ) {  
            saldo = saldo - retiro;  
        }  
    }  
  
    public void imprimir() {
```

```
        System.out.println( "La cuenta es de: " + nombre
                            + ", número: " + numero
                            + ". Es una cuenta de " + tipo
                            + ", con saldo: " + saldo + "\n");
    }
}
```

Ejercicio 2.5. Modificar el programa del ejercicio 2.3, teniendo en cuenta que ahora los atributos `saldo` y `numCuenta` son privados.

Solución:

```
public class Principal {
    public static main(String args[]) {
        Cuenta cuentaCredito;
        Cuenta cuentaDebito;

        // Creamos los objetos
        cuentaCredito = new Cuenta();
        cuentaDebito = new Cuenta();

        cuentaCredito.nombre = "Pedro Sanchez";
        cuentaCredito.setSaldo(1500.0);
        cuentaCredito.setNumero(244513);
        cuentaCredito.tipo = "crédito";

        cuentaDebito.nombre = "Pablo Garcia";
        cuentaDebito.setSaldo(7800.0);
        cuentaDebito.setNumero(273516);
        cuentaDebito.tipo = "débito";

        cuentaCredito.imprimir();
        cuentaDebito.imprimir();
    }
}
```

La palabra reservada `this` y los métodos "getters" y "setters"

En Java se utiliza la palabra reservada `this` para denotar desde el interior de un objeto una referencia al propio objeto. Si tenemos el objeto `pedro` de la clase `Persona`, entonces `pedro.this` es el mismo objeto `pedro`. `this` tiene varios usos y en esta sección veremos una de sus aplicaciones más comunes.

Cuando los atributos son privados, como en el caso de `saldo` y `numero`, entonces se codifican los métodos `getSaldo()` y `getNumero()` para que otros objetos tengan

acceso a su valor. A este tipo de métodos se les conoce como *getters*. Si además queremos que los otros objetos también puedan modificar el valor de estos atributos, entonces usamos los métodos *setters*, que en este caso son `setSaldo()` y `setNúmero()`:

```
public void setSaldo(double s) {  
    saldo = s;  
}  
  
public void setNúmero(int num) {  
    numero = num;  
}
```

Nótese que para diferenciar entre el valor enviado como parámetro y el nombre del atributo usamos `s` y `num`, respectivamente. Declarar los atributos como privados permite *encapsular* los datos del objeto.

El encapsulamiento sirve para proteger los datos de los objetos y se logra declarando los atributos de una clase como `private` y codificando métodos especiales para controlar el acceso. La manera de acceder a los atributos desde afuera de la clase es por medio de los métodos *getter* y la manera de modificar los atributos desde afuera de la clase es usando los métodos *setter*.

Como el encapsulamiento es una práctica común, muchos ambientes de desarrollo orientado a objetos codifican automáticamente los métodos *getters* y *setters* con las siguientes reglas:

1. Tanto *getters* como *setters* son públicos.
2. Los *getters* no reciben parámetros y el tipo de dato que regresan es el mismo que el del atributo correspondiente. Su nombre comienza con `get` seguido del nombre del atributo pero iniciando con mayúscula y regresan el valor del atributo. Por ejemplo:

```
public double getSaldo() {  
    return saldo;  
}  
  
public int getNúmero() {  
    return numero;  
}
```

3. Los *setters* reciben como parámetro el mismo tipo de dato que el del atributo. El nombre de los métodos *setters* se construye de forma análoga a la de los *getters*, pero iniciando con la palabra `set`, y asignan al atributo el valor del parámetro recibido. El parámetro recibido tiene el mismo nombre que el atributo. Para diferenciar entre el valor enviado como parámetro y el nombre

del atributo se utiliza la palabra `this`, de tal forma que `this.nombreAtributo` se refiere al atributo del objeto. Por ejemplo:

```
public void setSaldo( double saldo ) {
    // el parametro saldo se asigna al atributo this.saldo
    this.saldo = saldo;
}

public void setNumero( int numero ) {
    // el parametro numero se asigna al atributo
    // this.numero
    this.numero = numero;
}
```

En general, los atributos de una clase se declaran como privados y se accede a estos por medio de los métodos *setters* y *getters*.

Los métodos constructores

Un constructor es un método especial, que se invoca al momento de crear un objeto, es decir, cuando se instancia la clase. El método constructor *tiene exactamente el mismo nombre que el de la clase que construye*. Por ejemplo:

```
Reloj r = new Reloj();
```

Los métodos constructores tienen la tarea de dejar al objeto listo para su uso. Por ejemplo, en el reloj, el constructor debería por lo menos inicializar la hora a las 00:00:00. Además de la inicialización de variables (incluyendo el alojamiento de memoria dinámico para éstas), el constructor *debe encargarse de crear los objetos que forman parte del objeto*.

Todas las clases tienen un constructor por omisión, con el cuerpo vacío y sin parámetros. El constructor por omisión siempre existe y evita problemas cuando no se definen explícitamente constructores en una clase. Definamos un constructor para la clase `Cuenta`:

```
public class Cuenta {
    String nombre;
    double saldo;
    int numero;
    String tipo;

    public Cuenta( String n, double s, int num, String t ) {
        nombre = n;
```

Ahora, para construir un objeto de la clase `Cuenta` será necesario proporcionar los datos de la cuenta por medio de los parámetros del constructor. Por ejemplo, se crea el objeto `cuentaCredito`, desde la clase `Principal`, definiendo desde ese momento que dicha cuenta esté a nombre de Pedro Sánchez, con saldo de 1500, número de cuenta 244513 e indicando que es una cuenta de crédito. Otro objeto `cuentaDebito` puede crearse a nombre de Pablo García, con saldo de 7800, número de cuenta 273516 e indicando que es una cuenta de débito. Para lograr lo anterior, hacemos lo siguiente:

```
public class Principal {  
    public static void main(String[] args) {  
        Cuenta cuentaCredito;  
        Cuenta cuentaDebito;  
  
        // Creamos los objetos  
        cuentaCredito = new Cuenta( "Pedro Sanchez", 1500, 244513,  
                                    "crédito" );  
        cuentaDebito = new Cuenta( "Pablo Garcia", 7800, 273516,  
                                    "débito" );  
        ...  
    }  
}
```

this en los métodos constructores

La palabra reservada **this** también es útil para dar el mismo nombre de los atributos a los parámetros que recibe un método constructor. Enseguida presentamos el código del constructor de la clase **Cuenta** utilizando la palabra **this** para indicar que se trata del atributo de la clase:

```
public Cuenta( String nombre, double saldo, int numero,
               String tipo) {
    this.nombre = nombre;
    this.saldo = saldo;
    this.numero = numero;
    this.tipo = tipo;
}
```

Al utilizar **this.nombreAtributo**, evitamos las ambigüedades cuando los parámetros tienen el mismo nombre que los atributos.

Varios constructores en una misma clase

Es posible definir múltiples constructores para una misma clase; los cuales representan modos diferentes de inicializar un objeto. Se diferencian entre sí por los parámetros que reciben. A continuación presentamos la clase **Reloj**, la cual tiene un método constructor sin parámetros que inicializa los atributos con cero. Tiene además el método **visualizar()**, el cual despliega la hora en pantalla, el método **tic()** que incrementa la hora actual en un segundo y define el comportamiento del reloj cada vez que se oye un "tic" y el método **ponerALas()**, que pone el reloj a cierta hora, minuto y segundo, de acuerdo con los valores que recibe en sus parámetros:

```
public class Reloj {
    int h;
    int m;
    int s;

    // método constructor (se llama igual que la clase)
    public Reloj() {
        h = 0;
        m = 0;
        s = 0;
    }

    public void visualizar() {
        System.out.println( h + ":" + m + ":" + s );
    }

    public void tic() {
```

```

s++;
if ( s >= 60 ) {
    s = 0;
    m++;
    if ( m >= 60 ) {
        m = 0;
        h++;
        if ( h >= 12 ) {
            h = 0;
        }
    }
}

public void ponerALas( int hora, int min, int seg ) {
    h = hora;
    m = min;
    s = seg;
}
}

```

Ahora tenemos la clase principal `PruebaReloj`, en la que se crea un objeto de la clase `Reloj` y se hace uso de sus métodos:

```

public class PruebaReloj {
    public static void main( String[] args ) {
        Reloj r = new Reloj(); // El constructor no lleva parametros
        r.visualizar();
        r.ponerALas( 1, 21, 58 );
        r.tic();
        r.visualizar();
        r.tic();
        r.visualizar();
    }
}

```

Ejercicio 2.6. Determina, sin ver la solución, cuál será la salida del programa anterior.

Solución:

```

0:0:0
1:21:59
1:22:0

```

Ahora pondremos un constructor adicional en la clase `Reloj`, de tal forma que la clase tenga un constructor sin parámetros y otro con parámetros:

```
public class Reloj {  
    int h;  
    int m;  
    int s;  
  
    // metodo constructor (se llama igual que la clase)  
    public Reloj() {  
        h = 0;  
        m = 0;  
        s = 0;  
    }  
  
    // Este segundo constructor acepta parametros  
    public Reloj( int h, int m, int s ) {  
        this.h = h;  
        this.m = m;  
        this.s = s;  
    }  
  
    public void visualizar() {  
        System.out.println( h + ":" + m + ":" + s );  
    }  
  
    public void tic() {  
        s++;  
        if ( s >= 60 ) {  
            s = 0;  
            m++;  
            if ( m >= 60 ) {  
                m = 0;  
                h++;  
                if ( h >= 12 ) {  
                    h = 0;  
                }  
            }  
        }  
    }  
  
    public void ponerALas( int hora, int min, int seg ) {  
        h = hora;  
        m = min;  
        s = seg;  
    }  
}
```

En la clase `PruebaReloj` creamos dos relojes, `r1` y `r2`. Como no se usan parámetros al crear el objeto `r1`, la clase usa su primer constructor. Al crear `r2` se usa el segundo constructor debido a que se incluyen los parámetros:

```
public class PruebaReloj {  
    public static void main( String[] args ) {  
        // El constructor no lleva parametros  
        Reloj r1 = new Reloj();  
  
        //Se usa el constructor con parametros  
        Reloj r2 = new Reloj( 11, 24, 59);  
  
        r1.visualizar();  
        r1.tic();  
        r1.visualizar();  
  
        r2.tic();  
        r2.visualizar();  
    }  
}
```

Ejercicio 2.7. Determina, sin ver la solución, cuál será la salida del programa anterior.

Solución:

```
0:0:0  
0:0:1  
11:25:0
```

Objetos dentro de los objetos

La propiedad de construir objetos que contienen a su vez otros más es una de las características más destacadas de la POO, pues promueve la versatilidad en el diseño de las clases.

Agreguemos a nuestro ejemplo de la clase `Reloj` la clase `ContadorCiclico`, cuyos objetos se caracterizan por tener una cuenta actual y una cuenta máxima. Así, por ejemplo, el contador cíclico de un minutero tiene 60 como cuenta máxima, mientras que el de la hora tiene 12. El método `incrementar()` incrementa en uno la cuenta del contador cíclico y reinicia la cuenta si se llega a la cuenta máxima:

```
public class ContadorCiclico {  
    int cuenta;  
    int max;  
  
    //Constructor  
    public ContadorCiclico( int m) {  
        max = m;  
        cuenta = 0;  
    }
```

```
public void ponerEn( int c ) {
    cuenta = c;
}

public int incrementar() {
    cuenta = ( cuenta + 1 ) % max;
    return cuenta;
}

public String toString() {
    return Integer.toString( cuenta );
}
}
```

Modifiquemos la clase `Reloj` para que tenga contadores cíclicos. Ahora la clase `Reloj` tiene atributos que son a su vez objetos. Los atributos de un objeto, que son a su vez objetos, se crean normalmente en el método constructor, que es lo más adecuado para que el objeto que se está creando se pueda usar inmediatamente.

En el constructor de la clase `Reloj` se crean los contadores que marcan la hora (h), el minuto (m) y el segundo (s). Obsérvese que el constructor de la clase `ContadorCiclico` requiere como parámetro la cuenta máxima:

```
public class Reloj {
    ContadorCiclico h;
    ContadorCiclico m;
    ContadorCiclico s;

    public Reloj() {
        h = new ContadorCiclico( 12 );
        m = new ContadorCiclico( 60 );
        s = new ContadorCiclico( 60 );
    }

    public String toString() {
        return h + ":" + m + ":" + s;
    }

    public void tic() {
        if ( s.incrementar() == 0 ) {
            if ( m.incrementar() == 0 ) {
                h.incrementar();
            }
        }
    }

    public void ponerALas( int hora, int min, int seg ) {
```

Ahora, al pasar un segundo, el comportamiento de `Reloj` está definido de forma más clara.

```

        h.ponerEn( hora );
        m.ponerEn( min );
        s.ponerEn( seg );
    }
}

```

Recordemos el método anterior para incrementar la cuenta del reloj en un segundo:

```

public void tic() {
    s++;
    if ( s >= 60 ) {
        s = 0;
        m++;
        if ( m >= 60 ) {
            m = 0;
            h++;
            if ( h >= 12 ) {
                h = 0;
            }
        }
    }
}

```

y comparémoslo con el nuevo, que usa tres objetos de clase **ContadorCiclico**:

```

public void tic() {
    if ( s.incrementar() == 0 ) {
        if ( m.incrementar() == 0 ) {
            h.incrementar();
        }
    }
}

```

Este último es un claro ejemplo de la utilidad de la POO para hacer código más fácil de leer. La instrucción `s.incrementar()` invoca al método de la clase **ContadorCiclico** que incrementa la cuenta. Además, regresa un entero que indica la cuenta actual. Luego, la expresión `s.incrementar() == 0` compara la cuenta actual del segundero con cero: si es cero, entonces incrementa la cuenta del minutero que, a su vez, la compara con cero. La hora se incrementa únicamente cuando la cuenta del minutero y del segundero son cero. Con este nuevo código, el método `tic()` en `Reloj` no incluye el comportamiento de los contadores cíclicos, sino solamente el comportamiento de un `Reloj` *interactuando con sus contadores internos*. El comportamiento de los contadores cíclicos se codifica aparte.

Otra característica importante de los objetos es su facultad de desplegar su contenido en forma de texto. Esto se codifica en un método que se llama `toString()`, el cual siempre debe regresar una cadena de caracteres. Se dice que el método

`toString()` proporciona una representación del objeto en una cadena de caracteres, es decir, en un `String` o texto. Observemos el método `toString()` de la clase `reloj`:

```
public String toString() {  
    return h + ":" + m + ":" + s;  
}
```

Este método concatena los valores de la cuenta de cada contador cíclico de forma que da la hora, el minuto y el segundo separados por dos puntos, es decir, `h:m:s`.

La clase `ContadorCiclico` también tiene su representación en cadena de caracteres:

```
public String toString() {  
    return Integer.toString( cuenta );  
}
```

Hay que poner especial atención a lo siguiente. La instrucción

```
return h + ":" + m + ":" + s;
```

provoca que se *invoque automáticamente* el método `toString()` de la clase `ContadorCiclico`, ya que la concatenación (o suma) de un objeto de esta clase con una cadena de caracteres requiere su representación como cadena de caracteres. En general, cuando un objeto que tiene un método `toString()` es colocado donde se espera un `String`, el compilador Java lo substituye por un llamado a su método `toString()`, para colocar en su lugar el `String` que lo representa.

De igual manera, se invoca automáticamente el método `toString()` de la clase `Reloj` cada vez que se usa un objeto `Reloj` en donde debería haber un `String`. Por ejemplo, cuando se envía el objeto a pantalla con `System.out.println` en la clase principal:

```
public class PruebaReloj {  
    public static void main( String[] args ) {  
        Reloj r = new Reloj();  
        System.out.println( r );  
        r.ponerALas( 1, 21, 58 );  
        r.tic();  
        System.out.println( r );  
        r.tic();  
        System.out.println( r );  
    }  
}
```

Se requiere que el objeto `r` sea convertido a una cadena de caracteres.

A continuación se describe lo que sucede en la clase `PruebaReloj`:

- Se invoca al método `main()` de la clase `PruebaReloj`.
- En `main()` se crea el objeto `r` de clase `Reloj`, invocando al constructor de `Reloj`.
- El constructor de `Reloj` crea los tres contadores cíclicos, invocando al constructor de `ContadorCiclico`.
- El constructor de `ContadorCiclico` inicializa `max` y `cuenta`.

Véase la figura II-7 que muestra el objeto `r` con sus tres contadores. Cada contador tiene su cuenta máxima y está inicializado con la cuenta en cero:

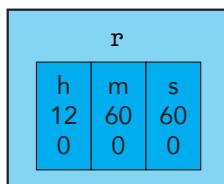


Figura II-7 El objeto `r` de clase `Reloj`

- Se manda imprimir `r` a la pantalla. Como se requiere un `String`, se invoca automáticamente `r.toString()` para usar su resultado en lugar de `r`.
- En `r.toString()` se invoca automáticamente `h.toString()`, `m.toString()` y `s.toString()`, los cuales regresan el valor de su cuenta en forma de `String`.
- Se pone el reloj a las 1:21:58. El método `PonerALas()` se invoca al método `ponerEn()` de cada contador.
- Se hace `tic()` y se escribe en pantalla el reloj dos veces.

Ejercicio 2.8. Determina, sin ver la solución, cuál será la salida del programa anterior.

Solución:

```
0:0:0
1:21:59
1:22:0
```

En UML, un objeto de la clase `Reloj`, que contiene tres objetos de clase `ContadorCiclico`, se modela como una relación llamada composición. De esta manera, el reloj está compuesto por tres contadores cíclicos que se crean dentro de la clase `Reloj`. Cuando desaparece el reloj, también desaparecen los tres contadores.

En la figura II-8 se ilustra, en UML, que el reloj está compuesto por tres contadores de la clase **ContadorCiclico**.

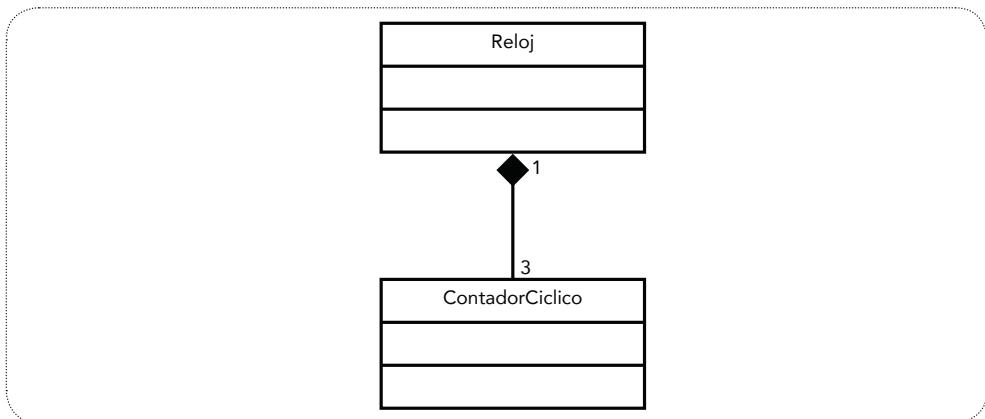


Figura II-8. Relación de composición: un reloj está compuesto por tres contadores cílicos

Wrappers

Java es un lenguaje orientado a objetos *casi* puro. Aparte de estos, existen las entidades que no son objetos. Son valores de tipo simple, es decir, números (enteros y reales), caracteres y booleanos. En el cuadro II-1 se ilustran los tipos de datos primitivos en Java. Para cada tipo de dato primitivo, en Java existe una clase correspondiente que representa el mismo tipo de dato en términos de objetos. A estas clases se les llama *wrappers*.

CUADRO II-1. TIPOS DE DATOS PRIMITIVOS EN JAVA

<i>Tipo de dato primitivo</i>	<i>Descripción</i>
bool	Valores booleanos, constantes true y false
char	16 bits
byte	8 bits (-128, ..., 127)
short	16 bits (-2 ¹⁵ , ..., 2 ¹⁵ -1)
int	32 bits (-2 ³¹ , ..., 2 ³¹ -1)
long	64 bits (-2 ⁶³ , ..., 2 ⁶³ -1)
float	32 bits (IEEE754)
double	64 bits (IEEE754)
void	

La clase wrapper de un dato primitivo representa el mismo tipo de dato pero en un estilo orientado a objetos y proporciona funcionalidad clásica para operar con el tipo de dato que representa. Enseguida presentamos la lista de clases wrapper para cada tipo primitivo. Nótese que los nombres de las clases wrappers comienzan con mayúscula.

```
bool → Boolean  
char → Character  
int → Integer  
long → Long  
float → Float  
double → Double  
void → Void
```

Así, por ejemplo, si tenemos:

```
Double precio;
```

`precio` se comporta como una variable de tipo `double`, pero también es un objeto y, por ejemplo, se obtiene su equivalente en cadena de caracteres, invocando uno de sus métodos de la siguiente forma: `precio.toString()`.

Definición de constantes

En Java, las constantes se definen como atributos con el descriptor `final` y, por ser constantes, se declaran `static`. La sintaxis es la siguiente:

```
static final <tipoConstante> <nombreConstante> = <valor>;
```

Por convención, los nombres de las constantes deben estar en mayúsculas, con las palabras separadas por subrayado “`_`”. Por ejemplo, declaramos la constante `PI` en la clase `Test2`, de la siguiente forma:

```
public class Test2 {  
    public static final double PI = 3.1415;  
    ...  
}
```

Para asignar a la variable `v` el valor de la constante `PI`:

```
double v = Test2.PI;
```

Cadenas de caracteres

En Java, las cadenas de caracteres son objetos de la clase **String**. Se tratan como objetos, pero sin cambios de estado, como valores constantes, es decir, el objeto no es modificable (deberían haberse llamado **ConstString**, pero quizás sea un nombre demasiado largo). Por lo tanto, las cadenas no deben ser vistas como contenedores de caracteres. En Java, la clase que permite manipular cadenas como contenedores de caracteres se llama **StringBuffer**. Los siguientes son ejemplos de cadenas de caracteres (**String**).

```
String nombre = "Sandra Alvarez";
String carrera = "Informatica";
String frase = nombre + " es un estudiante de la carrera de "
              + carrera;
```

Las constantes del tipo **String** se encierran entre comillas:

```
"hasta"
"luego"
```

El lenguaje Java proporciona un soporte particular para los objetos **String**, proporcionando el operador de concatenación **+** y también formas automáticas de conversión de tipos de datos primitivos en cadenas. Por ejemplo:

```
double valor = 17.5;
String st = "La temperatura es " + valor + " grados";
```

En el ejemplo anterior, **valor** se convierte automáticamente en **String** y se concatena a las otras dos cadenas. Así obtenemos la cadena final:

```
"La temperatura es 17.5 grados"
```

que se asigna a la variable **st**.

En Java, todos los objetos tienen una representación textual, que se obtiene invocando automáticamente al método **toString()** cuando es necesario. El método **toString()** pertenece a la clase **Object**, de la cual derivan todas las clases. En el siguiente ejemplo se invoca a **c.toString()** automáticamente cuando se usa **c** como parámetro de **println**:

```
Counter c = new Counter( 1 );
c.incrementar();
System.out.println( c );
```

Es posible definir una representación textual específica para los objetos de una clase *redefiniendo* el método `toString()`, por ejemplo:

```
class Persona {  
    private String nombre;  
    private Fecha fechaNacimiento;  
  
    public Persona( String n, Fecha fn ) {  
        nombre = n;  
        fechaNacimiento = new Fecha( fn );  
    }  
  
    public String toString() {  
        return nombre + " nació el: " + fechaNacimiento;  
    }  
}
```

Un error frecuente consiste en usar el operador `==` para probar la igualdad (`!=` para la desigualdad) entre dos cadenas. Es un error porque `==` compara las referencias a los objetos y no los propios objetos. La siguiente clase ilustra lo que sucede con diferentes tipos de comparaciones entre cadenas:

```
public class TestDeIgualdad {  
    public static void main( String[] args ) {  
        String s1 = "prueba";  
        String s2 = "prueba"; // se le asigna el mismo objeto  
                           // que a s1  
        String s3 = new String( "prueba" ); // otro objeto igual  
        boolean test1 = ( s1 == s2 ); // son el mismo objeto  
        boolean test2 = ( s1 == s3 ); // no son el mismo  
        boolean test3 = ( s1.equals( s2 ) ); // contienen lo mismo  
        boolean test4 = ( s1.equals( s3 ) ); // contienen lo mismo  
        System.out.println( test1 );  
        System.out.println( test2 );  
        System.out.println( test3 );  
        System.out.println( test4 );  
    }  
}
```

Ejercicio 2.9. Determina, sin ver la solución, cuál será la salida del programa anterior.

Solución:

```
true  
false  
true  
true
```

La figura II-9 ilustra la ubicación en la memoria de las cadenas de caracteres del código anterior:

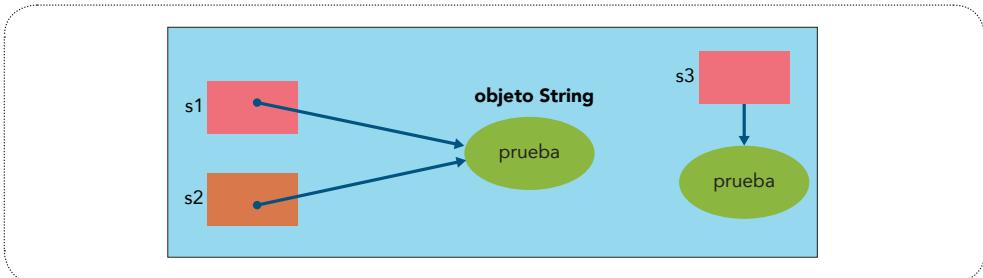


Figura II-9. Ubicación de las cadenas de caracteres `s1`, `s2` y `s3`

Las instrucciones

```
String s1 = "prueba";
String s2 = "prueba";
```

indican que tanto `s1` como `s2` apuntan al mismo objeto: "prueba". Sin embargo, con

```
String s3 = new String( "prueba" );
```

se aloja una nueva ubicación de memoria con el mismo contenido. Es por eso que con `test1 = (s1 == s2)` el resultado es verdadero y con `test2 = (s1 == s3)` el resultado es falso, ya que verificamos si `s1`, `s2` y `s3` están apuntando a la misma localidad y no si el contenido de lo que están apuntando es igual. Las instrucciones

```
test3 = ( s1.equals( s2 ) );
test4 = ( s1.equals( s3 ) );
```

sí comparan el contenido de las cadenas, por lo que en los dos casos la comparación resulta verdadera.

Ejercicios resueltos

Los siguientes ejercicios forman parte del caso de estudio de este libro, en el que crearemos diversas clases de objetos que modelan piezas de rejoles o relojes completos. En general, la POO es una forma de organizar el código de un programa de manera que los objetos sean semejantes a los objetos reales. En los siguientes capítulos usaremos las clases aquí presentadas para ilustrar conceptos más avanzados.

Ejercicio 1. Haremos la clase `Display`, la cual debe tener tres pares de dígitos como atributos que son los que el objeto debe mostrar en pantalla. Para ello pondremos como atributo privado un arreglo de tres `String` al que llamaremos `par`. Los métodos de `Display` son:

- El constructor `Display`, que crea los tres `String` del arreglo `par[]` y los inicializa en "00",
- El método `poner()`, que pone en el arreglo la representación en dos dígitos de tres números enteros,
- El método `toString()` para obtener la lectura del display de la siguiente forma: 00:00:00.

En la primera parte de la solución se presenta la estructura de la clase `Display`.

Primera solución parcial

```
public class Display {
    private String par[]; // arreglo de pares de digitos

    Aquí va el encabezado del metodo constructor {
        par = instanciar el arreglo con 3 pares de digitos
        poner( 0, 0, 0 ); // este método los pone en cero
    }

    public void poner( Integer num[] ) {
        // ¿Cómo se convierten los Integer a cadena de
        // caracteres?
    }

    public String toString() {
        ...
    }
}
```

En la segunda solución parcial está codificado el constructor de la clase.

Segunda solución parcial

```
public class Display {
    private String par[]; // arreglo de pares de digitos

    public Display() {
        par = new String[3];
        poner( 0, 0, 0 );
    }
}
```

```
public void poner( Integer num[ ] ) {  
    // ¿Como se convierten los Integer a cadena de  
    // caracteres?  
}  
  
public String toString() {  
    ...  
}  
}
```

Ahora especificaremos el comportamiento del método `poner()`:

- Recibe como parámetros tres `Integer` en el orden que se van a desplegar.
- Toma de estos `Integer` los dos dígitos menos significativos y descarta los demás. Se deshace del signo negativo si lo tiene.
- Convierte a `String` cada uno de los `Integer` y, si alguno es de un dígito, concatena un cero a la izquierda.

Recordemos que `Integer` es una clase wrapper, por lo que el método `toString()` de `Integer` convierte el valor entero a una cadena de caracteres. En la tercera solución parcial presentamos el código de `poner()`.

Tercera solución parcial

```
public class Display {  
    private String par[]; // arreglo de pares de digitos  
  
    public Display() {  
        par = new String[3];  
        poner( 0, 0, 0 );  
    }  
  
    public void poner( Integer num[ ] ) {  
        for ( int i = 0; i < 3; i++ ) {  
            num[i] = num[i] % 100; // para asegurar un numero valido  
            if ( num[i] < 0 ) {  
                num[i] = -num[i];  
            }  
            if ( num[i] < 10 ) {  
                par[i] = "0" + num[i];  
            }  
            else {  
                par[i] = num[i];  
            }  
        }  
    }  
}
```

```
public String toString() {  
    ...  
}  
}
```

Por último, el método `toString()` de `Display` convierte el objeto a una cadena de caracteres de la forma: `##:##:##`.

La solución final contiene todos los métodos de `Display` es la siguiente:

Solución final

```
public class Display {  
    private String par[]; // arreglo de pares de digitos  
  
    public Display() {  
        par = new String[3];  
        poner( 0, 0, 0 );  
    }  
  
    public void poner( Integer num[] ) {  
        for ( int i = 0; i < 3; i++ ) {  
            num[i] = num[i] % 100; // para asegurar un numero valido  
            if ( num[i] < 0 ) {  
                num[i] = -num[i];  
            }  
            if ( num[i] < 10 ) {  
                par[i] = "0" + num[i];  
            }  
            else {  
                par[i] = num[i];  
            }  
        }  
    }  
  
    public String toString() {  
        return par[0] + ":" + par[1] + ":" + par[2];  
    }  
}
```

Ejercicio 2. Escribiremos la clase `Manecilla` que tiene como atributos privados su `largo` y su `ancho` de tipo `Double` (para que pueda dibujarse a sí misma en pantalla con estas dimensiones y, aunque esta parte del código queda fuera del alcance de este libro, lo consideraremos para mostrar que estos atributos son parte natural de un objeto de este tipo) y también tiene el `numero` (del 1 al 12) y la `marca` (las que están

entre ese número y el siguiente y va del 0 al 4) hacia donde apunta. Los métodos de `Manecilla` son:

- El constructor `Manecilla`, que inicializa los cuatro atributos mencionados.
- El método `moverAPosicion()`, que mueve la manecilla a un `numero` y `marca` dados como parámetros, asegurándose que el `numero` y `marca` recibidos estén en el rango válido.
- El método `toString()` regresa en una sola cadena la posición de la manecilla.

En la primera solución parcial presentamos la estructura de la clase `Manecilla` con su método constructor codificado.

Primera solución parcial

```
public class Manecilla {  
    private Integer numero;  
    private Integer marca;  
    private Double largo;  
    private Double ancho;  
  
    public Manecilla( Double l, Double a, Integer n, Integer m ) {  
        largo = l;  
        ancho = a;  
        moverAPosicion( n, m );  
    }  
  
    public void moverAPosicion( Integer n, Integer m ) {  
        ...  
    }  
  
    public String toString() {  
        ...  
    }  
}
```

¿Cómo codificarías el método `moverAPosicion()` para asegurar que el número esté entre el 1 y el 12, y la marca entre el 0 y el 4? En la segunda solución parcial está codificado el este método.

Segunda solución parcial

```
public class Manecilla {  
    private Integer numero;
```

```
private Integer marca;
private Double largo;
private Double ancho;

public Manecilla( Double l, Double a, Integer n, Integer m ) {
    largo = l;
    ancho = a;
    moverAPosicion( n, m );
}

public void moverAPosicion( Integer n, Integer m ) {
    numero = n % 12; // del 0 al 11 y despues haremos que el 0
                      // se "vea" como 12
    marca = m % 5;
}

public String toString() {
    ...
}
```

Finalmente, codificaremos el método `toString()` para que se pueda mostrar en pantalla la posición de la manecilla en relación a los números del reloj en forma de texto. A continuación se presenta la solución final.

Solución final

```
public class Manecilla {
    private Integer numero;
    private Integer marca;
    private Double largo;
    private Double ancho;

    public Manecilla( Double l, Double a, Integer n, Integer m ) {
        largo = l;
        ancho = a;
        moverAPosicion( n, m );
    }

    public void moverAPosicion( Integer n, Integer m ) {
        numero = n % 12; // del 0 al 11 y despues haremos que el 0
                          // se "vea" como 12
        marca = m % 5;
    }

    public String toString() {
        Integer n = numero;
        if ( n == 0 ) {
            n = 12; // para que el 0 se "vea" como 12
        }
        ...
    }
}
```

```
    }
    return n + "/" + marca
           + " largo:" + largo + " ancho:" + ancho;
}
}
```

Ejercicio 3. Haremos la clase **ContadorCiclico** (una ligeramente diferente de la que se hizo en “Objetos dentro de los objetos”) con las siguientes características:

Un **ContadorCiclico** tiene como atributos privados su **cuenta** (de tipo **Integer**) y el número máximo **max** al que puede llegar a contar (de tipo **Integer**).

Los métodos de **ContadorCiclico** son:

- El constructor **ContadorCiclico()**, que inicializa la cuenta en un número dado como parámetro y establece la cuenta máxima posible según un segundo parámetro.
- **incrementar()** incrementa en uno la **cuenta** y cuando se alcanza la cuenta máxima ésta regresa a cero.
- **getCuenta()**, para obtener el valor actual de la cuenta (regresa un **Integer**).
- **getMax()**, para obtener el valor de **max** (regresa un **Integer**).

En la primera solución parcial, se presenta la estructura general de la clase **ContadorCiclico**.

Primera solución parcial

```
public class ContadorCiclico {
    private Integer cuenta;
    private Integer max;

    public ContadorCiclico( Integer c, Integer m ) {
        ...
    }

    public Integer incrementar() {
        ...
    }

    public Integer getCuenta() {
        ...
    }

    public Integer getMax() {
```

```
    ...
}
```

Para codificar su constructor hay que considerar que éste inicializa la cuenta con el parámetro `c` y, por medio del módulo, garantiza que no se pase de la cuenta máxima `m`, también recibida como parámetro. En la segunda solución parcial se codifica el constructor de la clase `ContadorCiclico`.

Segunda solución parcial

```
public class ContadorCiclico {
    private Integer cuenta;
    private Integer max;

    public ContadorCiclico( Integer c, Integer m ) {
        cuenta = c % m;
        max = m;
    }

    public Integer incrementar() {
        ...
    }

    public Integer getCuenta() {
        ...
    }

    public Integer getMax() {
        ...
    }
}
```

Una vez iniciada la cuenta de `ContadorCiclico`, ésta no se puede alterar, sólo se incrementa por medio del método `incrementar()`. Codificaremos este método de tal forma que se incremente la cuenta pero que no se pase de la cuenta máxima.

Tercera solución parcial

```
public class ContadorCiclico {
    private Integer cuenta;
    private Integer max;

    public ContadorCiclico( Integer c, Integer m ) {
        cuenta = c % m;
        max = m;
    }
```

```
public Integer incrementar() {  
    cuenta = ( cuenta + 1 ) % max;  
    return cuenta;  
}  
  
public Integer getCuenta() {  
    ...  
}  
  
public Integer getMax() {  
    ...  
}  
}
```

Finalmente, codificaremos los métodos `getCuenta()` y `getMax()`.

Solución final

```
public class ContadorCiclico {  
    private Integer cuenta;  
    private Integer max;  
  
    public ContadorCiclico( Integer c, Integer m ) {  
        cuenta = c % m;  
        max = m;  
    }  
  
    public Integer incrementar() {  
        cuenta = ( cuenta + 1 ) % max;  
        return cuenta;  
    }  
  
    public Integer getCuenta() {  
        return cuenta;  
    }  
  
    public Integer getMax() {  
        return max;  
    }  
}
```

Prácticas de laboratorio

El objetivo de las prácticas de laboratorio que aquí se proponen es aplicar los conceptos estudiados en este capítulo. Se plantean dos prácticas, las cuales se retomarán en los capítulos subsecuentes para incorporar los temas que ahí se expongan.

Robot

Se requiere controlar un robot que existe dentro de un espacio bidimensional discreto, como el que se muestra en la figura II-10.

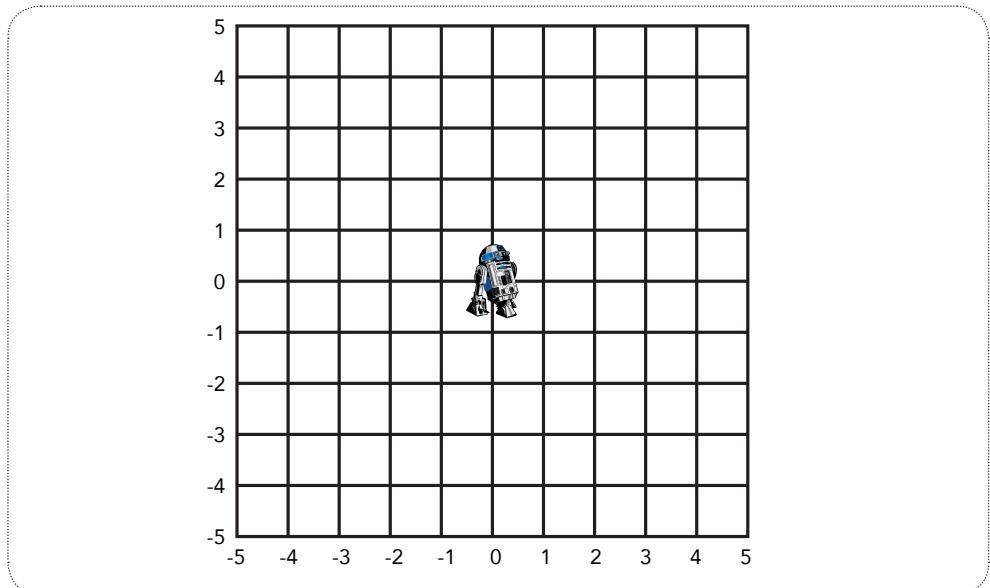


Figura II-10. Un robot que existe en un espacio bidimensional discreto

Este robot puede hacer giros de 90 grados en sentido contrario a las manecillas del reloj, uno a la vez, y avanzar hacia el frente una unidad en cada desplazamiento. Para lograr esto, se está considerando la clase `Robot`, cuyos atributos, constructores y métodos se describen a continuación.

Atributos:

- `x`, `y`, de tipo Integer, que representan la posición horizontal y vertical, respectivamente, del robot dentro del espacio bidimensional discreto.
- `direccion`, de tipo Integer, para representar, en grados, la dirección que tiene el robot, que puede ser 0, 90, 180 o 270.

Constructores:

- `Robot()`, constructor que establece los valores por omisión 0, 0 y 0 para los atributos `x`, `y` y `direccion`, respectivamente.
- `Robot(int xi, int yi, int dir)`, constructor que establece los valores en los parámetros `xi`, `yi` y `dir` para los atributos `x`, `y` y `direccion`, respectivamente.

Métodos adicionales a los *getters* y *setters*:

- **girar()**, que indica al robot que tiene que hacer un giro de 90 grados en sentido contrario a las manecillas del reloj.
- **avanzar()** hace que el robot avance una posición en la dirección que el robot tiene actualmente.

Implementa la clase **Robot** considerando los atributos, constructores y métodos descritos. Además, escribe una clase principal para crear un par de **Robots** y hacer que giren y avancen y, en cada movimiento, saber cuál es su ubicación y su dirección.

Tienda virtual

Una persona desea tener una tienda virtual para vender libros y películas en formato *blu-ray*. Para esto es necesario, primeramente, representar con un enfoque orientado a objetos, los tipos de productos que se van a vender. Inicialmente, se están considerando las dos clases siguientes:

- **Libro**, que tiene como atributos el autor y el título de tipo **String**, y el precio de tipo **Float**.
- **Pelicula**, cuyos atributos son el **título**, el **protagonista** y el **director** de tipo **String**, y el precio de tipo **Float**.

Las dos clases contienen sus respectivos constructores, métodos *getters* y *setters* y el método **toString()**.

Implementa las clases **Libro** y **Pelicula** considerando los atributos, constructores y métodos descritos. Además, escribe una clase principal para crear varios **Libros** y varias **Películas**, y saber su título, autor, protagonista, director y precio, según corresponda.

Cuestionario

Contesta las siguientes preguntas:

1. ¿Cuál es la diferencia entre una clase y un objeto?
2. ¿Cuál es la función de un método constructor?
3. ¿En que sistema operativo se puede correr un programa en Java?
4. ¿Por qué es bueno el encapsulamiento?
5. ¿Cómo se logra el encapsulamiento en Java?
6. ¿Qué es una clase *wrapper*?
7. ¿Cuál es el nombre del método utilizado para que los atributos de un objeto se impriman en forma de cadena de caracteres?

Relaciones entre clases

Objetivo

Comprender el significado de las relaciones de *generalización*, *agregación* y *asociación* en la POO y saber cómo se implementan en el lenguaje de programación Java.

Introducción

Como los objetos no existen aisladamente, es muy importante estudiar las relaciones que existen entre sí. Aquí estudiamos las relaciones básicas. La decisión para establecer una u otra relación entre dos objetos no es una tarea de programación, sino de diseño orientado a objetos y depende básicamente de la experiencia del diseñador.

Utilizaremos UML para representar las relaciones entre clases. En la figura III-1 se muestra un glosario de notación UML para elaborar diagramas de clases.

Glosario de notaciones para diagramas de clases				
Símbolo	Significado			
<table border="1"><tr><td>Clases</td></tr><tr><td>Atributos</td></tr><tr><td>Métodos</td></tr></table>	Clases	Atributos	Métodos	Clase
Clases				
Atributos				
Métodos				
<table border="1"><tr><td><<interface>></td></tr><tr><td>operaciones</td></tr></table>	<<interface>>	operaciones	interfaz	
<<interface>>				
operaciones				
→	Generalización			
.....→	Implementación de interfaz			
— 1 1...*	Asociación Asociación con multiplicidad			
—→	Asociación con navegabilidad			
—○	Agregación			
—◆	Composición			

Figura III-1. Glosario de notaciones UML para diagramas de clases

Los símbolos de la figura III-1 se utilizan para elaborar diagramas como el del ejemplo de la figura III-2.

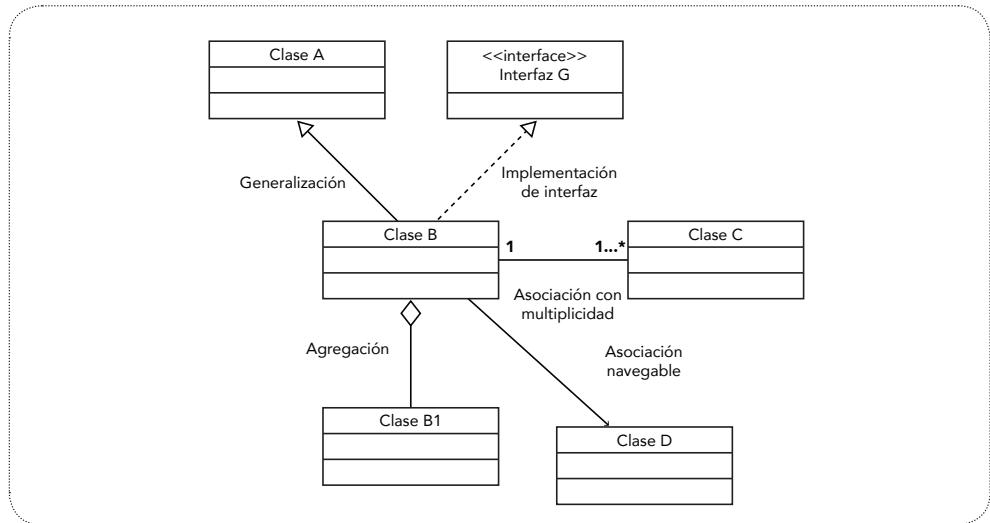


Figura III-2. Ejemplo que ilustra el uso de la notación en la construcción de un diagrama de clases

Ahora revisaremos en qué consisten las relaciones de generalización, asociación, agregación y composición. En el ultimo capítulo (el Quinto) se estudian las interfaces.

La generalización (herencia)

La generalización es una relación entre clases en las que hay una clase padre, llamada superclase, y una o más clases hijas especializadas, a las que se les denomina *subclases*. La herencia es el mecanismo mediante el cual se implementa la relación de generalización. En la práctica, cuando se codifica un sistema, se habla de herencia en lugar de generalización.

Cuando hay herencia, la clase hija o subclase adquiere los atributos y métodos de la clase padre. Como se ilustra en la figura III-3, hay herencia cuando existe la relación *es un* entre los objetos de una clase hijo y una madre. Por ejemplo: un **Profesor** *es un* **Empleado**, un **Administrativo** *es un* **Empleado**, un **Empleado** *es una* **Persona**, un **Estudiante** *es una* **Persona**.

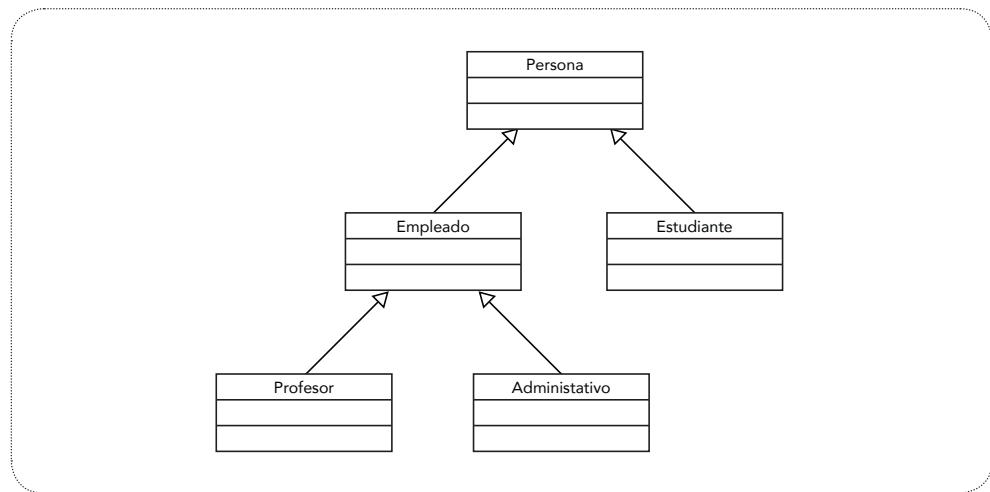


Figura III-3. La herencia se representa con la relación es un, por ejemplo, un **Profesor** es un **Empleado**

Los métodos creados en la clase hija tienen acceso tanto a los métodos, como a los atributos públicos de la clase padre (véanse los ejemplos más adelante).

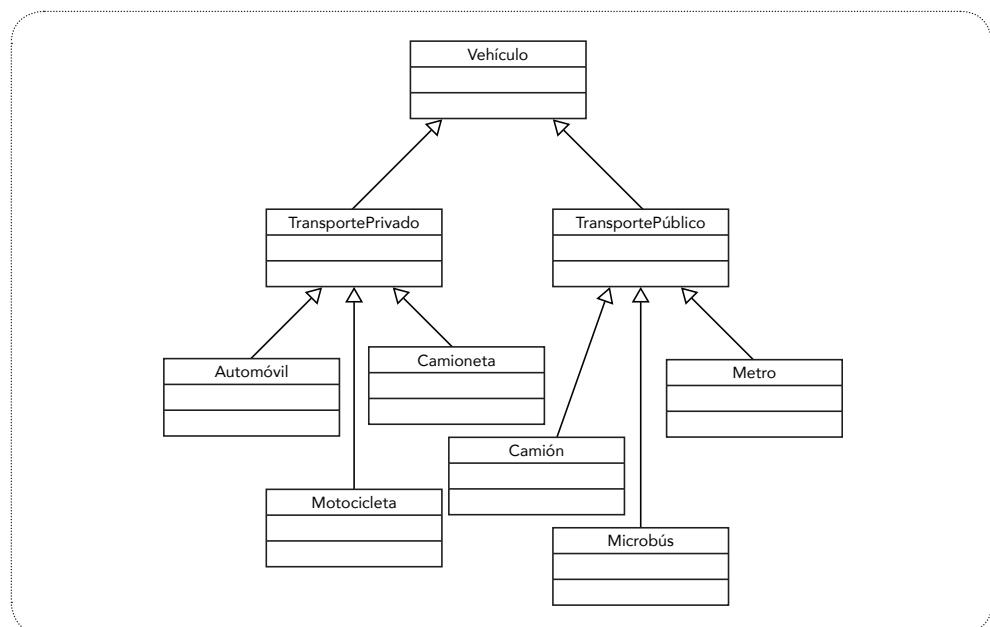


Figura III-4 La relación es un es válida para hijos, nietos, etc., por ejemplo, una **Motocicleta** es un **Vehículo**

La figura III-4 muestra otro ejemplo de jerarquía de herencia: la **Motocicleta** es un **TransportePrivado** y a la vez es un **Vehículo**, ya que todo **TransportePrivado** es un **Vehículo**.

tePrivado es un **Vehículo**. Lo mismo puede decirse del **Automóvil** y la **Camioneta**. El **Camión** es un **TransportePublico** y también es un **Vehículo**, lo mismo que el **Microbús** y el **Metro**.

Si la clase **Vehículo** tiene los métodos públicos **ponerEnMarcha()**, **acelerar()** y **detener()**, entonces todos los objetos descendientes de **Vehículo**, es decir, **Automóvil**, **Camión**, etc., pueden usar estos métodos.

Generalización y especialización

La generalización y la especialización son dos perspectivas diferentes de la misma relación de herencia. Con la *generalización* se buscan clases que sean de nivel superior a alguna clase en particular. En la figura III-5 se muestran tres ejemplos de herencia, en los que, vistos de abajo hacia arriba, hay una relación de *generalización*. Sin embargo, vistos de arriba hacia abajo, hay una relación de *especialización*.

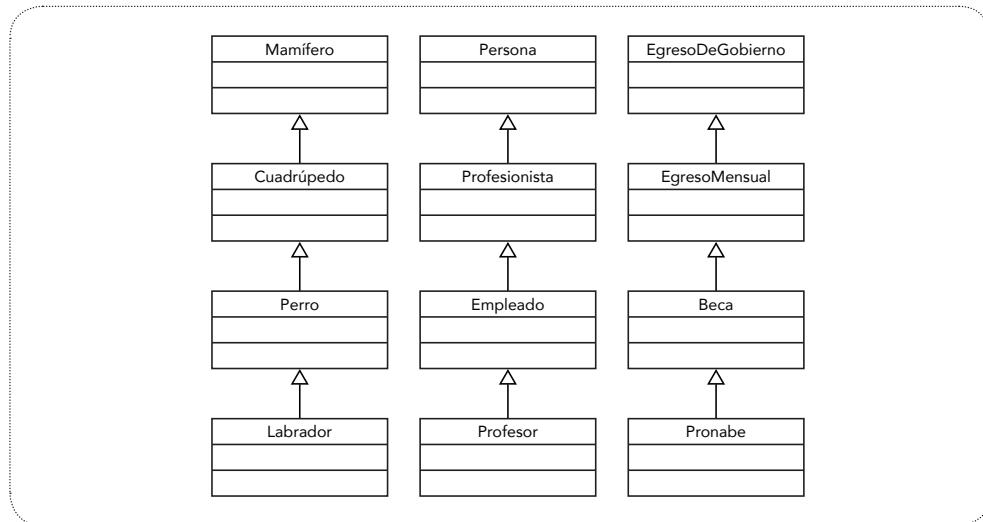


Figura III-5. Ejemplos de generalización

Con la *generalización* se buscan propiedades comunes entre varios objetos similares que puedan agruparse para formar una nueva clase genérica. En el ejemplo, se piensa en que un **Labrador** tiene propiedades comunes con otros "objetos", como pueden ser un **Boxer** o un **Dálmata**, que se pueden agrupar en una superclase llamada **Perro**.

A la contraparte de la *generalización* se le llama *especialización* y consiste en encontrar subclases de nuestra clase actual, en las cuales se detallan las propiedades particulares de cada una de ellas. Una subclase representa la *especialización* de la clase superior (superclase).

Todos los **Empleados** tienen atributos comunes, por ejemplo, su nombre, dirección, número de empleado, etcétera, y métodos comunes como **calcularQuincena()**, **calcularAntiguedad()**, etc. Sin embargo, hay cosas que caracterizan sólo a los informáticos, por ejemplo, las técnicas y herramientas de software que manejan. Así, se va construyendo una relación jerárquica de herencia entre los objetos, buscando todas las subclases que podría tener alguna clase. En la figura III-6 se ilustra la jerarquía obtenida en este ejemplo.

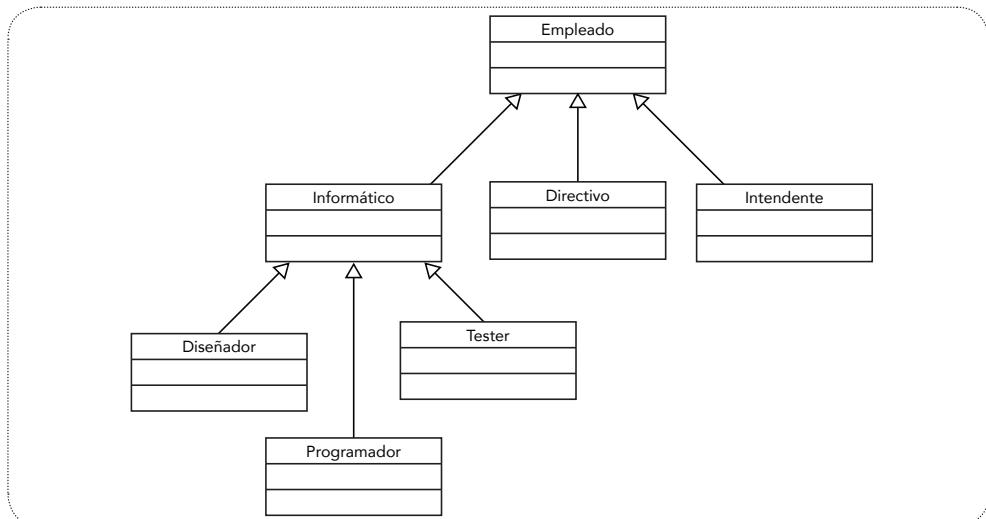


Figura III-6. Ejemplo de especialización

Existencia de la relación de herencia

La clase de la cual se hereda es la clase *padre*, a la que también se le llama *superclase*. La clase a la cual se hereda es la clase *hija*, a la que también se le llama *subclase*. Por ejemplo, la clase padre **Persona** puede tener dos hijas: **Alumno** y **Profesor**. En este caso, **Persona** es la superclase de las clases **Alumno** y **Profesor**, las cuales se dice que son subclases de **Persona**. Con el mecanismo de herencia, las clases hijas heredan el mismo código que contiene su clase padre, es decir, tienen los mismos métodos y atributos del padre.

De esta manera, implementamos la *generalización*, ya que las clases hijas no tienen que repetir ese código. Además, se agregan métodos y atributos particulares para cada subclase y así implementamos la *especialización*, ya que este código no pertenece a la clase padre. Conviene verificar que exista la relación *es un* para diseñar una jerarquía de clases. Si “B” no *es un* “A”, entonces “B” no debería heredar de “A”. En el primer ejemplo de la figura III-7 se propone que una **Pila** *es una* **Lista** y que una **Cola** también *es una* **Lista**, por lo que **Pila** y **Cola** heredarían todas las

propiedades de **Lista**. Esto no es correcto, ya que las operaciones que se hacen con una **Pila** no incluyen todas las operaciones que pueden hacerse con una **Lista**. Sería extraño, por ejemplo, que a una **Pila** se le pudiera agregar un elemento en cualquier posición y no en el tope como es normal. Entonces, aunque para implementar una **Pila** sea común usar una **Lista**, una **Pila** no es una **Lista**, sino que hace uso de una.

En el segundo ejemplo de la figura III-7 no existe la relación de herencia, ya que un país no es un continente.

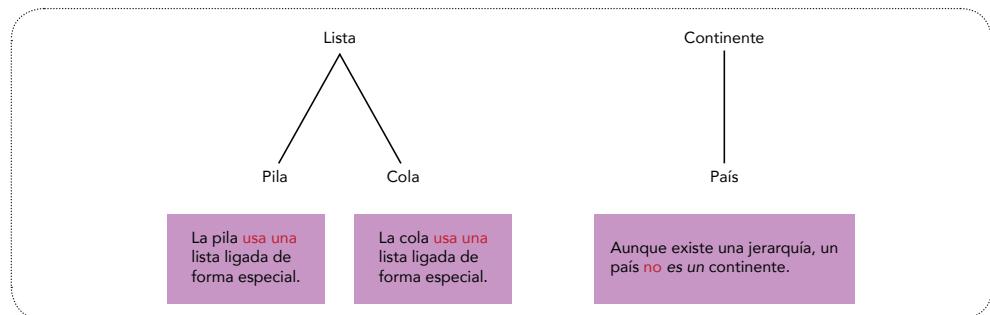


Figura III-7. La relación de herencia sólo existe si la clase hija es un tipo de la clase padre. En estos ejemplos no hay herencia

Para establecer una relación de herencia en el segundo caso, podemos, por ejemplo, declarar como padre a **Territorio** y como hijos a **Continente** y **País**. De esta forma sí se cumple la relación *es un Territorio* para las dos clases.

La herencia facilita que un hijo se comporte según sus características específicas, sin tener que codificar nuevamente todos los métodos que ya contiene el padre. Si se requiere, por ejemplo, implementar la clase **Continente**, se reutiliza fácilmente el código hecho para la clase **Territorio**, pero con algunas restricciones y adiciones que la conviertan en un **Continente** sin dejar de ser un **Territorio**. Un **Continente** es un tipo especial de **Territorio** y hereda de éste sus propiedades.

Cuando definimos una clase hija, indicamos de alguna manera cuál es su clase padre y luego definimos los atributos y métodos adicionales propios de la clase hija. En Java, la relación de herencia se especifica con la palabra reservada **extends**, lo que significa que la definición de una clase extiende la definición de la superclase.

Implementando la herencia

Hagamos una clase llamada **Cronometro**, la cual contiene todos los métodos y atributos de la clase **Reloj**, pero además contiene el atributo booleano **pausa** y el

método `pausarSeguir()`, el cual cambia de estado el atributo `pausa`: lo pone en `verdadero` si está en `falso` y en `falso` si está en `verdadero`. Esto con la idea de que un `Cronometro` es *un Reloj* especial que puede ser detenido y echado a andar en cualquier momento.

Será necesario inicializar `pausa` en `falso` en el constructor de `Cronometro`. Además, habrá que redefinir el método `tic()` para que, imitando el funcionamiento del botón de pausa-sigue de un cronómetro, se incremente la cuenta únicamente cuando `pausa` esté en `falso`, es decir, cuando no está oprimido el botón de pausa.

Recordemos la clase `Reloj`:

```
public class Reloj {  
    ContadorCiclico h;  
    ContadorCiclico m;  
    ContadorCiclico s;  
  
    public Reloj() {  
        h = new ContadorCiclico( 0,12 );  
        m = new ContadorCiclico( 0,60 );  
        s = new ContadorCiclico( 0,60 );  
    }  
  
    public String toString() {  
        return h + ":" + m + ":" + s;  
    }  
  
    public void tic() {  
        if ( s.incrementar() == 0 ) {  
            if ( m.incrementar() == 0 ) {  
                h.incrementar();  
            }  
        }  
    }  
  
    public void ponerALas( int hora, int min, int seg ) {  
        h.ponerEn( hora );  
        m.ponerEn( min );  
        s.ponerEn( seg );  
    }  
}
```

La clase `Cronometro` es hija de `Reloj`. Esto lo indicamos con la palabra reservada `extends`, ya que el `Cronometro` es una extensión de lo que es un `Reloj`. Codificaremos primero el constructor de la clase `Cronometro`.

```
public class Cronometro extends Reloj {
    Boolean pausa;

    Cronometro() {
        super();
        pausa = false;
    }

    public void pausarSeguir() {
        ...
    }

    public void tic() {
        ...
    }
}
```

super llama al constructor de la clase padre para crear al cronómetro primero como **Reloj**. Luego se continua con lo que se requiera adicionalmente para construir un cronómetro.

El método **pausarSeguir()** únicamente cambia el estado del atributo **pausa** cada vez que es llamado. Este método se ve como una forma de implementar el botón que tienen los cronómetros para pausar y reanudar. El método **tic()** substituye al que está en **Reloj**, ya que ambos tienen el mismo nombre y lista de parámetros. Este método se encarga de reaccionar a **tic()**, considerando el estado del cronómetro: si el **Cronometro** está en pausa, no hace nada; pero en caso contrario, hace lo mismo que un **Reloj** normal.

```
public class Cronometro extends Reloj {
    Boolean pausa;

    Cronometro() {
        super();
        pausa = false;
    }

    public void pausarSeguir() {
        pausa = !pausa;
    }

    public void tic() {
        if ( !pausa ) {
            super.tic();
        }
    }
}
```

Se redefine el método **tic()** para que haga lo mismo que el **tic()** de **Reloj** sólo cuando **pausa** está en **false**.

En un **Cronometro**, los métodos **visualizar()** y **ponerALas()** funcionan como están definidos en **Reloj** y por esto no se tienen que escribir de nuevo.

Ahora crearemos en la clase principal un objeto de clase Cronometro al cual llamaremos c. Como podemos observar en el siguiente código, c también tiene los atributos h, m y s porque también es un Reloj:

```
public class PruebaReloj {  
    public static void main( String[] args ) {  
        Cronometro c = new Cronometro();  
        System.out.println( c );  
        c.ponerALas( 1, 21, 58 );  
        System.out.println( c );  
        c.tic();  
        System.out.println( c );  
        c.pausarSeguir(); // se pone en pausa el cronometro  
        c.tic(); // no tiene efecto  
        System.out.println( c ); // se ve sin cambios  
        c.pausarSeguir(); // quitamos la pausa  
        c.tic();  
        System.out.println( c );  
    }  
}
```

Determina la salida del programa antes de ver la solución.

Solución:

```
0:0:0  
1:21:58  
1:21:59  
1:21:59  
1:22:0
```

Las clases abstractas

En el modelado orientado a objetos a veces sirve introducir clases de cierto nivel que incluso pueden no existir en la realidad, pero que su concepto es útil para organizar mejor la estructura de un programa. Estas clases se conocen como *clases abstractas*.

No se pueden instanciar objetos de una clase abstracta, sin embargo, si se pueden instanciar objetos de las subclases que heredan de una clase abstracta, siempre y cuando no se definan como abstractas.

Por ejemplo, si la clase Vehiculo de la figura III-4 es una clase abstracta, no podremos crear objetos de esta clase. Sin embargo, sí podemos crear objetos de sus hijas, por ejemplo, de Metro, de Motocicleta, de Automovil, etc. En la figura III-8 presentamos la jerarquía de clases del "Kit de Herramientas de Ventana Abstracta"

(AWT, del inglés Abstract Windowing Toolkit) de Java. El AWT suministra un sistema de ventanas para hacer una interfaz de usuario.

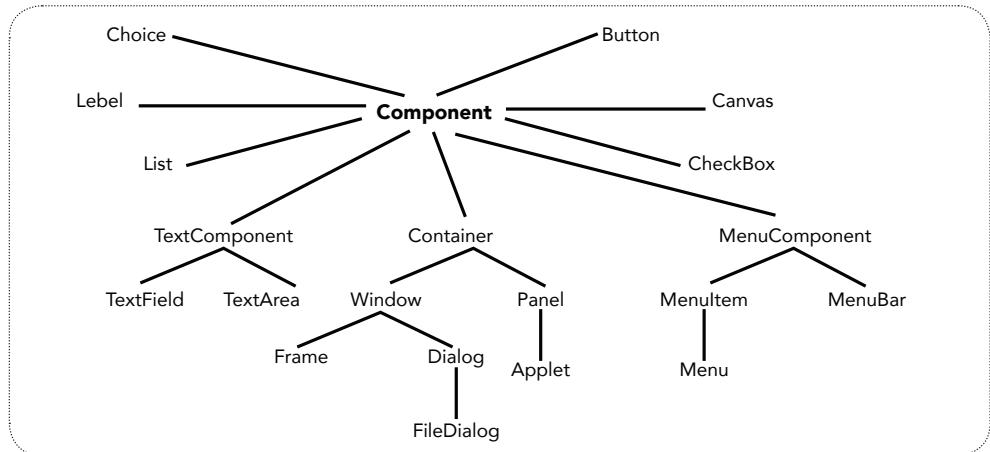


Figura III-8. Kit de Herramientas de Ventana Abstracta

La clase **Component** es una clase abstracta de la cual se derivan todas las demás. En el siguiente código ilustramos las clases de las cuales se pueden crear objetos y de las que no.

```

import java.awt.*; // paquete de herramientas para hacer ventanas

public class PruebaClasesAbstractas {
    public static void main( String[] args ) {
        // se puede crear un objeto de la clase Container
        Container c = new Container();

        // se puede crear un objeto de la clase Button
        Button boton = new Button( "Run" );

        Component c1 = new Component();
    }
}
  
```

Se redefine el método `tic()` para que haga lo mismo que el `tic()` de Reloj sólo cuando `pausa` está en `false`.

Una clase abstracta es como un “embrión” que tiene algunas partes vitales, como hígado, corazón y pulmones, sin embargo, no está listo para sobrevivir, necesita pasar por otras etapas de formación hasta tener todas sus partes vitales para poder crear objetos a partir de ella.

Ejercicios de herencia

Ejercicio 1. Supongamos que tenemos las clases **Alumno** y **Profesor**. Usando la generalización, podemos definir una clase padre que sea la clase **Persona**. La ventaja de hacerlo así es que reutilizamos el código de la clase **Persona** en las clases hijas. Además, el código relacionado con alumnos y con profesores queda por separado, lo que facilita el mantenimiento del programa. La figura III-9 ilustra esta relación de herencia.

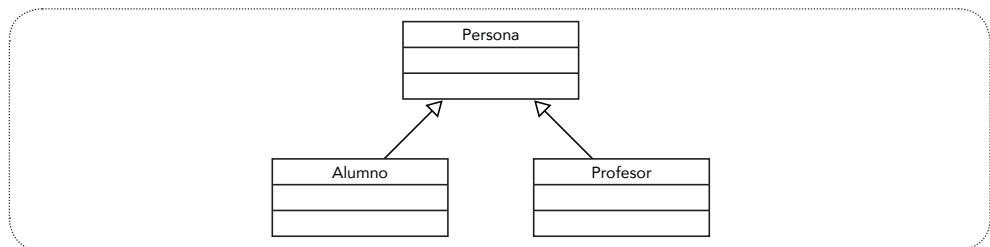


Figura III-9. Relación de herencia entre Persona, Alumno y Profesor

La clase **Persona** define las propiedades comunes. Queremos hacer la clase **Persona** con los atributos privados **nombre** de clase **String** y **fechaNacimiento** de clase **Fecha**. El constructor de **Persona** inicializa los atributos con un **nombre** (**String**) y un objeto de clase **Fecha** que recibe como parámetros:

```

public class Persona {
    private String nombre;
    private Fecha fechaNacimiento;

    public Persona( String n, Fecha fn ) {
        // para copiar un objeto se debe hacer una nueva instancia
        nombre = new String( n );
        fechaNacimiento = new Fecha( fn ); // ver constructor de
                                         // Fecha abajo
    }

    public String toString() {
        return nombre + " nacido el: " + fechaNacimiento;
    }
}
  
```

Nótese que los objetos **nombre** y **fechaNacimiento** sólo existen hasta que los creamos en el constructor.

Ahora es necesario definir la clase **Fecha** con los atributos privados **año**, **mes** y **día** de clase **Integer**. **Fecha** tiene dos constructores: uno inicializa los atributos con tres **Integer** (año, mes y día) que se le envían como parámetros y el otro recibe un objeto de tipo **Fecha**.

Los métodos de `Fecha` son: `toString()`, para regresar la cadena de caracteres con el día, el mes y el año, y `asignar()`, el cual es un método público que asigna un día, un mes y un año específicos al objeto. ¿Cómo codificarías estos métodos?

Solución parcial

```
public class Fecha {  
    private Integer anio;  
    private Integer mes;  
    private Integer dia;  
  
    static final String[] mesLetra = { "",  
                                      "Ene", "Feb", "Mar", "Abr", "May", "Jun",  
                                      "Jul", "Ago", "Sep", "Oct", "Nov", "Dic"  
    };  
  
    public Fecha( Integer a, Integer m, Integer d ) {  
        asignar( a, m, d );  
    }  
    public Fecha( Fecha f ) {  
        asignar( f.anio, f.mes, f.dia );  
    }  
  
    public void asignar( Integer a, Integer m, Integer d ) {  
        ...  
    }  
  
    public String toString() {  
        ...  
    }  
}
```

Solución final

```
public class Fecha {  
    private Integer anio;  
    private Integer mes;  
    private Integer dia;  
  
    static final String[] mesLetra = { "",  
                                      "Ene", "Feb", "Mar", "Abr", "May", "Jun",  
                                      "Jul", "Ago", "Sep", "Oct", "Nov", "Dic"  
    };  
  
    Fecha( Integer a, Integer m, Integer d ) {  
        asignar( a, m, d );  
    }  
  
    Fecha( Fecha f ) {
```

```

        asignar( f.anio, f.mes, f.dia );
    }

    public void asignar( Integer a, Integer m, Integer d ) {
        anio = a;
        mes = m;
        dia = d;
    }

    public String toString() {
        return dia + " de " + mesLetra[mes] + " de " + anio;
    }
}

```

Ejercicio 2. Una vez teniendo la clase **Fecha** que se utiliza en **Persona**, procedemos a definir la clase **Alumno**.

La clase **Alumno** debe tener las mismas propiedades que **Persona**, más algunas otras que no toda persona tiene pero que todo alumno sí. Para no reescribir las propiedades de la persona cuando se define la clase **Alumno**, se indica que la clase **Alumno** hereda de **Persona**. **Alumno** tiene además el atributo privado **matrícula** de tipo **String**.

En el constructor **Alumno()** se deben inicializar los atributos **nombre**, **fecha-Nacimiento** y la **matrícula**. Sin embargo, el constructor de **Persona** ya inicializa los primeros dos. Para llamar al constructor de la clase padre se utiliza la palabra **super** y se le envían como parámetros los datos necesarios. De esta forma, sólo será necesario inicializar el atributo adicional de **Alumno**, que es la **matrícula**.

Solución parcial

```

public class Alumno extends Persona {
    private String matricula;

    public Alumno( String n, Fecha f, String m ) {
        super( n, f );
        matricula = new String( m );
    }

    public String toString() {
        ...
    }
}

```

Para llamar al constructor de la super-clase se usa
super(<listaParametros>)

¿Cómo se define el método **toString()** de **Alumno** invocando al de **Persona** para concatenar la **matrícula** al resultado de **toString()** de **Persona**?

Solución final

```
public class Alumno extends Persona {
    private String matricula;

    public Alumno( String n, Fecha f, String m ) {
        super( n, f );
        matricula = new String( m );
    }

    public String toString() {
        return super.toString()
            + " mat: " + matricula;
    }
}
```

Para invocar un método de la superclase se usa
super.metodo()

Ejercicio 3. Ahora definamos la clase **Profesor**, la cual hereda también de **Persona** pero añade propiedades que sólo tiene el profesor. **Profesor** tiene el atributo privado **claveEmpleado** de tipo **Integer**. En el constructor **Profesor()** se deben inicializar los atributos **nombre**, **fechaNacimiento** y **claveEmpleado**.

Como el constructor de **Persona** ya inicializa los primeros dos, llamamos al constructor de la clase padre con los datos necesarios y entonces sólo inicializamos el atributo adicional de **Profesor** que es **claveEmpleado**.

Solución

```
public class Profesor extends Persona {
    private Integer claveEmpleado;

    public Profesor( String n, Fecha f, Integer c ) {
        super( n, f );
        claveEmpleado = new Integer( c );
    }

    public String toString() {
        return super.toString()
            + " clave: " + claveEmpleado;
    }
}
```

Para invocar un método de la superclase se usa
super.metodo()

Finalmente crearemos, desde la clase principal el objeto **elAlumno** de clase **Alumno** y el objeto **elProfe** de clase **Profesor**.

Los datos de `elAlumno` son los siguientes:

```
nombre: Eloy Mata Arce
fechaNacimiento: 1 feb 1990
matricula: 2008112233
```

Los datos de `elProfe` son los siguientes:

```
nombre: Elmer Homero Petatero
fechaNacimiento: 11 dic 1987
claveEmpleado: 23245
```

En la primera solución parcial presentamos la estructura de la clase principal.

Primera solución parcial

```
public class PersonasMain {
    public static void main( String[] args ) {
        Fecha f = new Fecha( 1990, 2, 1 );
        String n;
        n = "Eloy Mata Arce";

        // crear alumno
        ...
        // creaar profesor
        ...
        // imprimir los dos objetos
        ...
    }
}
```

Primero instanciamos un objeto de clase `Fecha` y le enviamos sus datos en el constructor

Para crear a `elAlumno` enviamos a su constructor el nombre de la persona como `String`, el objeto `f` de clase `Fecha` y, como se trata de un alumno, su matrícula también como `String`. Para crear a `elProfe` no es necesario hacer otro objeto `Fecha`, ya que podemos usar el mismo objeto `f` asignándole una nueva fecha.

Segunda solución parcial

```
public class PersonasMain {
    public static void main( String[] args ) {
        Fecha f = new Fecha( 1990, 2, 1 );
        String n;
        n = "Eloy Mata Arce";
```

```
// crear alumno
Alumno elAlumno = new Alumno(n, f, "2008112233");

// crear profesor
f.asignar(1987, 12, 11);
...

// imprimir los dos objetos
...
}

}
```

Cada vez que se requiere que un objeto se convierta a **String**, se ejecuta automáticamente su método **toString()**, el cual lo convierte a una representación en forma de **String**.

```
public class PersonasMain {
    public static void main( String[] args ) {
        Fecha f = new Fecha( 1990, 2, 1 );
        String n;
        n = "Eloy Mata Arce";

        // crear alumno
        Alumno elAlumno = new Alumno( n, f, "2008112233" );

        // crear profesor
        f.asignar( 1987, 12, 11 );
        n = "Elmer Homero Petatero";
        Profesor elProfe = new Profesor( n, f, 23245 );

        // imprimir los dos objetos
        System.out.println( elAlumno );
        System.out.println( elProfe );
    }
}
```

Determina la salida de este programa antes de ver la solución.
Después de la ejecución, veremos lo siguiente en pantalla:

```
Eloy Mata Arce nacido el: 1 de Feb de 1990 mat :2008112233
Elmer Homero Petatero nacido el: 11 de Dic de 1987 clave: 23245
```

Ejercicio 4. Como la clase **Persona** no es abstracta, también podemos incluir en la clase **PersonasMain** un tercer objeto de clase **Persona**, como se muestra a continuación:

```

public class PersonasMain {
    public static void main( String[] args ) {
        Fecha f = new Fecha( 1990, 2, 1 );
        String n;
        n = "Eloy Mata Arce";

        // crear alumno
        Alumno elAlumno = new Alumno( n, f, "2008112233" );

        // crear profesor
        f.asignar( 1987, 12, 11 );
        n = "Elmer Homero Petatero";
        Profesor elProfe = new Profesor( n, f, 23245 );

        // crear persona
        f.asignar( 1975, 09, 28 );
        n = "Tercera persona";
        Persona unaPersona = new Persona( n, f );

        // imprimir los objetos
        System.out.println( elAlumno );
        System.out.println( elProfe );
        System.out.println( unaPersona );
    }
}

```

El constructor sólo requiere dos parámetros.

Ejercicio 4.a. Determina, sin ver la solución, cuál será la salida del programa anterior.

Solución:

Eloy Mata Arce nacido el: 1 de Feb de 1990 mat: 2008112233
 Elmer Homero Petatero nacido el: 11 de Dic de 1987 clave: 23245
 Tercera persona nacido el: 28 de Sep de 1975

Pero si declaramos **Persona** como clase abstracta:

```

public abstract class Persona {
    private String nombre;
    private Fecha fechaNacimiento;

    public Persona( String n, Fecha fn ) {
        nombre = n;
        fechaNacimiento = new Fecha( fn );
    }

    public String toString() {
        return nombre + " nacido el: " + fechaNacimiento;
    }
}

```

entonces el compilador indicará un error al instanciar un objeto de la clase **Persona**.

Ejercicio 5. ¿Qué relaciones de herencia hay entre las clases de la figura III-10? ¿Cómo las organizarías?

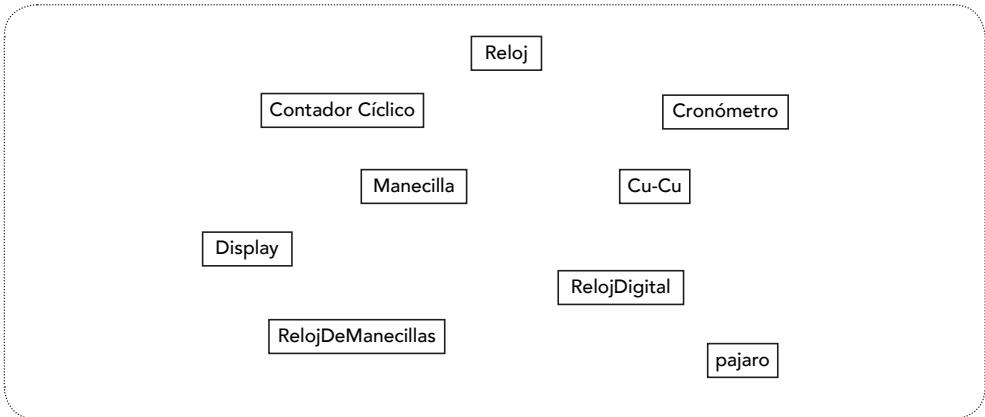


Figura III-10. Encontrar relaciones de herencia entre estas clases

Solución:

En la figura III-11 se ilustra la relación entre algunas de las clases de la figura III-10. Como se observa, no todas están relacionadas, pues las clases **ContadorCílico**, **Pájaro**, **Manecilla** y **Display** no tienen relación de herencia con otras clases.

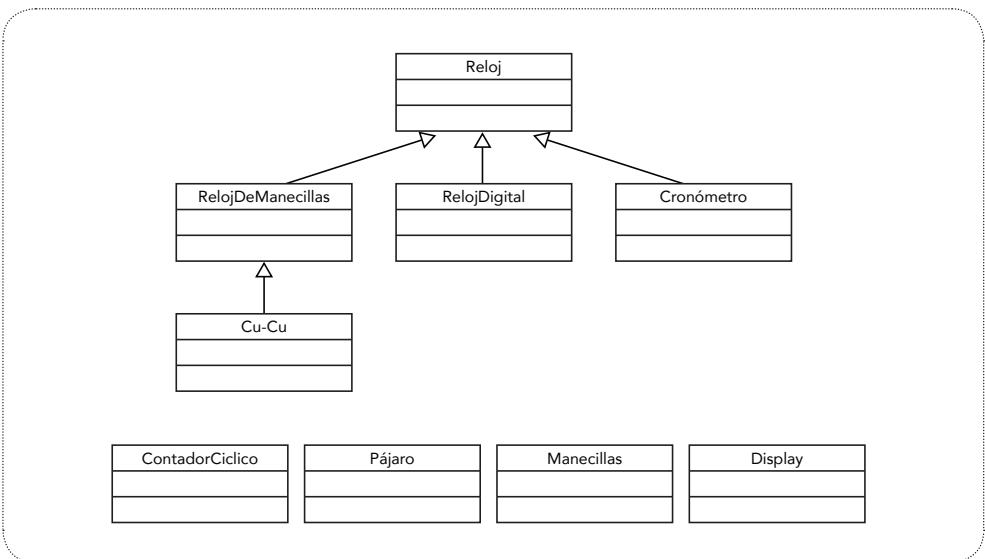


Figura III-11. No todas las clases tienen relaciones de herencia entre sí

En la figura III-11, la clase Reloj es una clase abstracta, ya que no se puede pensar en un objeto que sea solamente un reloj, pues en todo reloj se necesitan elementos adicionales para desplegar la hora (como manecillas, display, etc.). En esta figura observamos que existen cuatro tipos de relojes: el de manecillas, el cu-cú, el reloj digital y el cronómetro. A su vez, se considera al cu-cú como un reloj de manecillas.

Niveles de asociación

Se dice que un objeto de clase A está *asociado* con uno de clase B cuando el objeto de clase A hace uso de algún método del objeto de clase B. Existen tres niveles de asociación entre dos clases: la asociación simple, la agregación y la composición. En la figura III-12 se ilustran estos tres niveles, que se explican en cada una de las siguientes subsecciones.

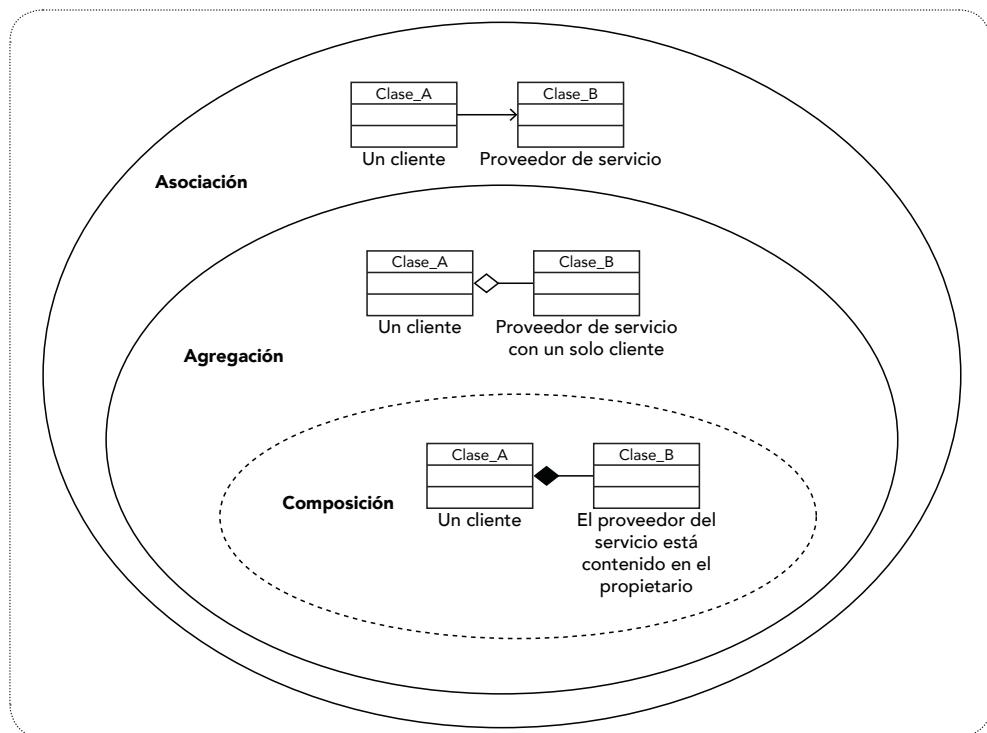


Figura III-12. Diagrama de Venn con los tres niveles de las relaciones de asociación

La asociación simple

Cuando una clase B está asociada a otra clase A se entiende que un objeto de clase B da servicio a otro de clase A. Este servicio puede exclusivo o compartido con otros

objetos de la clase A o de alguna otra clase a la que se asocie la clase B. Cada uno de los objetos es independiente del otro en cuanto a su creación o destrucción, es decir, si uno se crea o se destruye, el otro puede estar creado o no. La duración de la relación entre estos objetos es temporal, ya que la asociación puede crearse y destruirse en tiempo de ejecución sin que los objetos tengan que ser destruidos.

Por ejemplo, cuando se asigna un empleado para trabajar en dos proyectos. En este caso, el objeto empleado ya existe desde antes de que los dos objetos de clase **Proyecto** existan y seguirá existiendo aún cuando uno, o los dos proyectos terminen. Cada proyecto hace uso de las habilidades y conocimiento del empleado mientras el proyecto lo necesite.

En la figura III-13 se muestra la representación en UML de la relación de asociación entre las clases **Empresa**, **Proyecto** y **Empleado**. Un objeto de clase **Proyecto** y un objeto de clase **Empresa** pueden necesitar cualquier cantidad de objetos de clase **Empleado** y un objeto de clase **Empleado** puede estar asignado a uno o a varios objetos de clase **Proyecto** y a uno o a varios objetos de clase **Empresa**. Ejemplos: un empleado que trabaja en una empresa y también participa en proyectos fuera de su empresa, o un empleado que no tiene un trabajo fijo en una empresa pero que participa en varios proyectos, o un empleado que trabaja para dos empresas, etcétera.

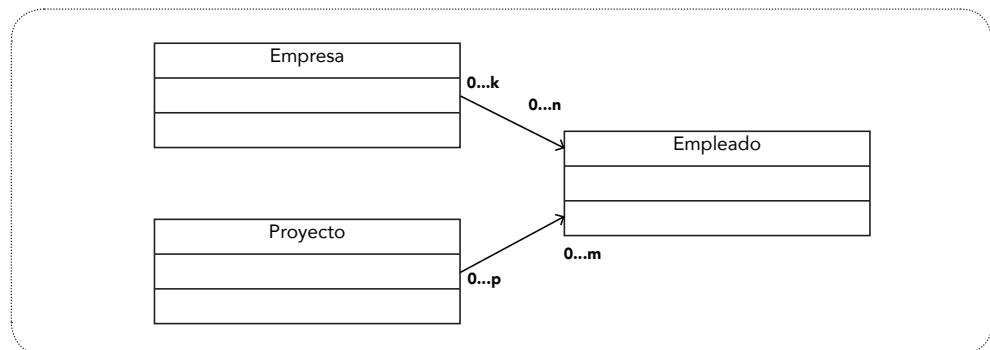


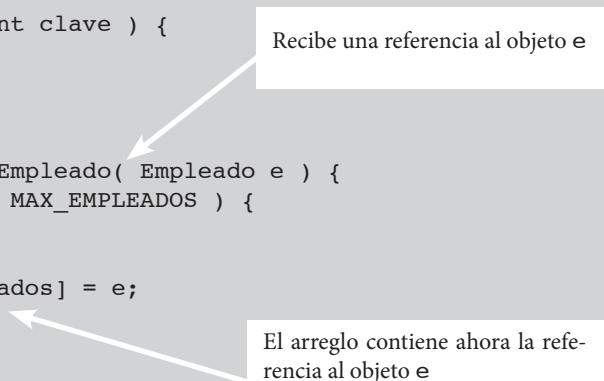
Figura III-13. A una **Empresa** se le asocian objetos **Empleado**, y a un **Proyecto** también se le asocian objetos **Empleado**

La agregación

Cuando una clase B está *agregada* a una clase A se entiende que uno o varios objetos de la clase B le dan servicio exclusivo a un objeto de la clase A. En este caso, cada objeto de clase B agregado a uno de clase A puede reasignarse a otro objeto también de clase A. Si el objeto de clase A desaparece, el objeto de clase B puede seguir existiendo, y éste debe agregarse a otro objeto de clase A para que su existencia tenga sentido.

La agregación de un objeto de clase B a un objeto de clase A se hace mediante un método de la clase A, el cual recibe como parámetro la referencia al objeto que se le va a agregar, que ya fue creado previamente. Por ejemplo, a un objeto que se llama `deptoVentas` de la clase `Departamento` se le agregan varios objetos de la clase `Empleado` por medio del método `agregarEmpleado` que está en la clase `Departamento`:

```
public class Departamento {  
    private static final int MAX_EMPLEADOS = 100;  
    private int clave;  
    private int numEmpleados;  
    private Empleado[] integrantes = new Empleado[MAX_EMPLEADOS];  
  
    public Departamento( int clave ) {  
        this.clave = clave;  
        numEmpleados = 0;  
    }  
  
    public Boolean agregarEmpleado( Empleado e ) {  
        if ( numEmpleados >= MAX_EMPLEADOS ) {  
            return false;  
        }  
        integrantes[numEmpleados] = e;  
        numEmpl++;  
        return true:  
    }  
  
    // los demás métodos de Departamento  
    ...  
}
```



El empleado `e` se crea previamente independientemente de la clase `Departamento`. Sin embargo, un empleado no tiene razón de ser si no pertenece a un departamento. Por lo tanto, si el departamento desaparece, aunque los empleados no desaparezcan, éstos deben ser reasignados a otro departamento.

En la figura III-14 se muestra la representación en UML de la relación de agregación entre las clases `Empleado` y `Departamento`. A cada `Departamento` se le agregan hasta cien objetos de la clase `Empleado`.

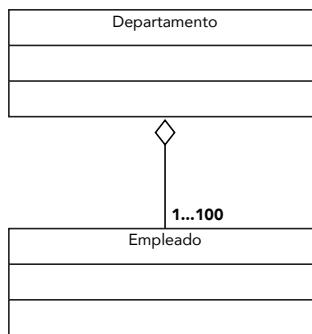


Figura III-14. Agregación de Empleado a Departamento

La composición

En una relación de *composición* entre A y B, en la que los objetos de la clase A tienen como componentes uno o más objetos de clase B. Los objetos de clase B son dependientes de la clase A ya que no pueden existir sin ser componentes de un objeto de clase A. Así, cuando desaparece el objeto de clase A, desaparecen también los objetos de clase B, porque no tiene sentido la existencia de B sin el objeto del que son componentes.

En el siguiente ejemplo, un cliente puede tener hasta tres cuentas, sin embargo estas cuentas no pueden existir si no existe ese cliente en particular, es decir, no pueden ser reasignadas a otro cliente. Con esta relación, cuando desaparece el cliente desaparecen también las cuentas asociadas al cliente:

```

Public class Cliente {
    private int clave;
    private String nombre ;
    private int cont;
    private Cuenta[] cuentas = new Cuenta[3];

    public Cliente( int clave, String nombre ) {
        this.clave = clave;
        this.nombre = nombre;
        cont = 0;
    }

    public Boolean agregarCuenta() {
        if ( cont >= 3 ) {
            return false;
        }
        cuentas[cont] = new Cuenta( this.clave );
    }
}
  
```

La cuenta se instancia dentro de la clase Cliente

```
cont++;
return true;
}

// los demás métodos de Cliente
...
}
```

En la implementación de la *composición*, la creación de un objeto de la clase **Cuenta** (`new Cuenta(...)`) se hace dentro del método `agregarCuenta(...)` que está en la clase **Cliente**. Cuando se borra un objeto **Cliente**, se eliminan también los objetos **Cuenta** que éste tiene.

En la figura III-15 se muestra la representación en UML de la relación de composición entre las clases **Cliente** y **Cuenta**. El diagrama dice que la clase **Cuenta** es un componente de la clase **Cliente** y que cada objeto **Cliente** puede estar compuesto de hasta tres objetos **Cuenta**.

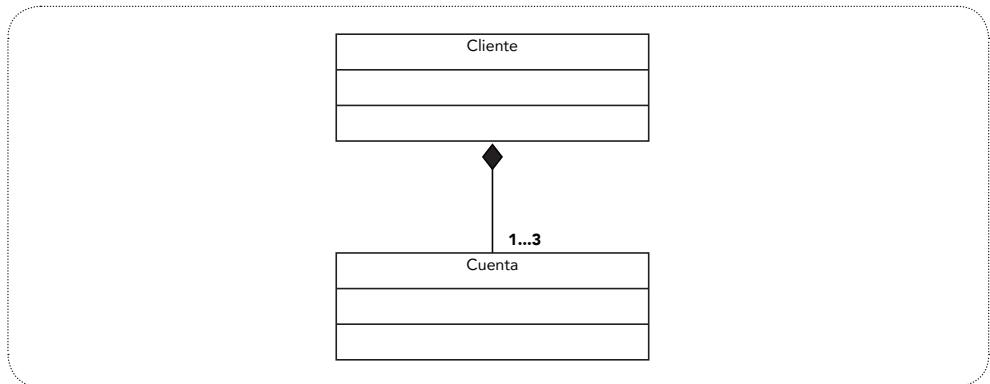


Figura III-15. Relación de composición entre un **Cliente** y sus **Cuentas**

Nótese la diferencia con la *agregación*, en la cual un empleado ya creado se pasa como referencia al método `agregarEmpleado(Empleado e)` de la clase **Departamento**. Si se borra un objeto de clase **Departamento**, no desaparecen los objetos agregados de clase **Empleado**. Sin embargo, en la *composición*, si se borra el objeto de clase **Cliente**, se borran también todas las cuentas que tiene como componentes.

Más ejemplos de composición

En la figura III-16 se muestran más ejemplos de la relación de composición. El **Display** es parte del **Cronómetro** y también de un **RelojDigital** (cada uno tiene

su **Display**). La **Manecilla** es parte del **RelojDeManecillas** y, finalmente, el **Pájaro** es parte del reloj **Cu-Cu**.

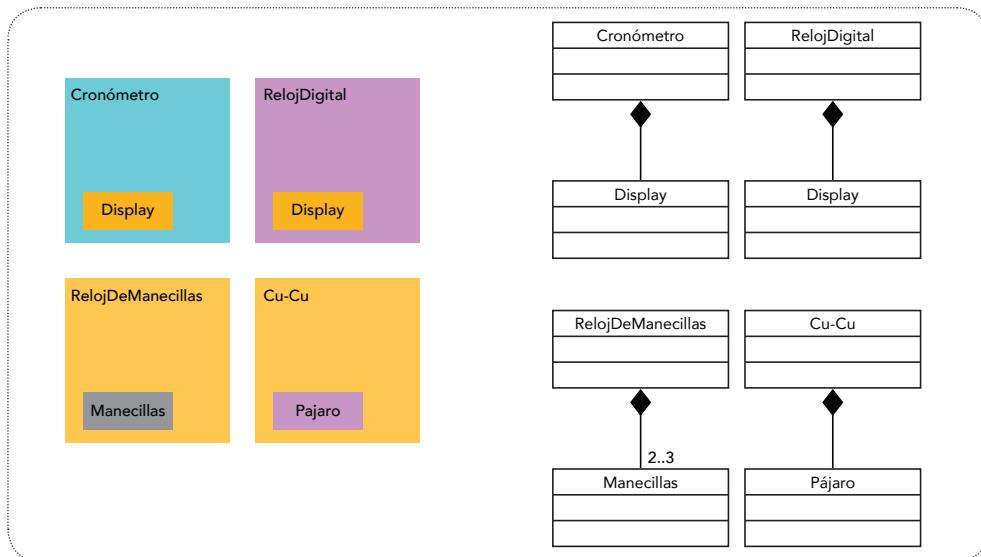


Figura III-16. Ejemplos de la relación de composición

Obsérvese que un reloj Cu-Cu también tiene manecillas, pero al ser éste un hijo (o descendiente) de **RelojDeManecillas**, se sabe que ya las tiene heredadas.

Ejercicios resueltos

Ejercicio 1. Implementar la clase **Reloj**, que es la clase padre de todos los relojes, según la figura III-11. **Reloj** es una clase abstracta con las siguientes características:

Tiene como atributos privados:

- Tres objetos de clase **ContadorCiclico** a los que llamaremos **hora**, **minuto** y **segundo**.
- Un indicador **am** de tipo booleano. Cuando **am** es verdadero indica que la hora es *antes meridiano* (a. m.) y cuando es falso indica que es *pasado meridiano* (p. m.).

Recordemos la clase **ContadorCiclico**:

```
public class ContadorCiclico {
    private Integer cuenta;
    private Integer max;
```

```
public ContadorCiclico( Integer c, Integer m ) {  
    cuenta = c % m;  
    max = m;  
}  
  
public Integer incrementar() {  
    cuenta = ( cuenta + 1 ) % max;  
    return cuenta;  
}  
  
public Integer getCuenta() {  
    return cuenta;  
}  
  
public Integer getMax() {  
    return max;  
}  
}
```

Los métodos de Reloj son:

- El constructor Reloj inicializa la hora, minuto y segundo. El atributo am se inicializa en función de la hora. Si la hora es menor que 12, entonces am es true.
- El método ponerAlas() lo usan tanto el constructor como los objetos externos y sirve para poner el reloj en una cierta hora, minuto y segundo.
- El método tic() hace avanzar un segundo la hora actual (en teoría este método se invocaría una vez cada segundo).
- Tres métodos que regresan un Integer para obtener la cuenta de la hora, el minuto y el segundo, respectivamente, y otro que regresa un Boolean para saber si es a. m. o no.
- En la primera solución parcial se presenta el esqueleto de la clase Reloj. En este caso, el constructor llama al método ponerAlas().

Primera solución parcial

```
public abstract class Reloj {  
    private ContadorCiclico hora;  
    private ContadorCiclico minuto;  
    private ContadorCiclico segundo;  
    private Boolean am;  
  
    public Reloj( Integer h, Integer m, Integer s ) {  
        ponerAlas( h, m, s );  
    }  
}
```

```
public void ponerALas( Integer h, Integer m, Integer s ) {  
    ...  
}  
  
public void tic() {  
    ...  
}  
  
public Integer getHora() {  
    ...  
}  
  
public Integer getMinuto() {  
    ...  
}  
  
public Integer getSegundo() {  
    ...  
}  
  
public Boolean getAm() {  
    ...  
}  
}
```

En la segunda solución parcial se presenta la implementación de los métodos `ponerALas()` y `tic()`. Nótese que en el método `ponerALas()` se deben modificar los valores de los contadores, pero, como la implementación de la clase `ContadorCiclico` no tiene un método para hacer esto, es necesario crear nuevos objetos cuyo valor inicial es el que se pide.

Segunda solución parcial

```
public abstract class Reloj {  
    private ContadorCiclico hora;  
    private ContadorCiclico minuto;  
    private ContadorCiclico segundo;  
    private Boolean am;  
  
    public Reloj( Integer h, Integer m, Integer s ) {  
        ponerALas( h, m, s );  
    }  
  
    public void ponerALas( Integer h, Integer m, Integer s ) {  
        hora = new ContadorCiclico( h, 12 );  
        minuto = new ContadorCiclico( m, 60 );  
        segundo = new ContadorCiclico( s, 60 );  
        am = h < 12;  
    }  
}
```

```
}

public void tic() {
    if ( segundo.incrementar() == 0 ) {
        if ( minuto.incrementar() == 0 ) {
            if ( hora.incrementar() == 0 ) {
                am = !am;
            }
        }
    }
}

public Integer getHora() {
    ...
}

public Integer getMinuto() {
    ...
}

public Integer getSegundo() {
    ...
}

public Boolean getAm() {
    ...
}
}
```

La solución final muestra también los métodos *getters*.

Solución final

```
public abstract class Reloj {
    private ContadorCiclico hora;
    private ContadorCiclico minuto;
    private ContadorCiclico segundo;
    private Boolean am;

    public Reloj( Integer h, Integer m, Integer s ) {
        ponerALas( h, m, s );
    }

    public void ponerALas( Integer h, Integer m, Integer s ) {
        hora = new ContadorCiclico( h, 12 );
        minuto = new ContadorCiclico( m, 60 );
        segundo = new ContadorCiclico( s, 60 );
        am = h < 12;
    }
}
```

```
public void tic() {
    if ( segundo.incrementar() == 0 ) {
        if ( minuto.incrementar() == 0 ) {
            if ( hora.incrementar() == 0 ) {
                am = !am;
            }
        }
    }
}

public Integer getHora() {
    return hora.getCuenta();
}

public Integer getMinuto() {
    return minuto.getCuenta();
}

public Integer getSegundo() {
    return segundo.getCuenta();
}

public Boolean getAm() {
    return am;
}
}
```

Ejercicio 2. Implementar la clase **RelojDeManecillas**, como descendiente de la clase abstracta **Reloj**, con las siguientes características:

Sus atributos privados son tres objetos de clase **Manecilla**: **horario**, **minutero** y **segundero**.

Los métodos de RelojDeManecillas son:

- El constructor **RelojDeManecillas** que crea las tres manecillas, una para la hora, otra para el minuto y otra para el segundo.
- El método **toString()** que, con base en el valor actual de las manecillas regresa una cadena de caracteres con el siguiente formato:

Manec: h:##/# m:##/# s:##/#

Como observamos en la figura III-17, **RelojDeManecillas** tiene todas las propiedades de un **Reloj**, además de que construye sus tres manecillas y, con **toString()**, da la lectura de las tres manecillas.

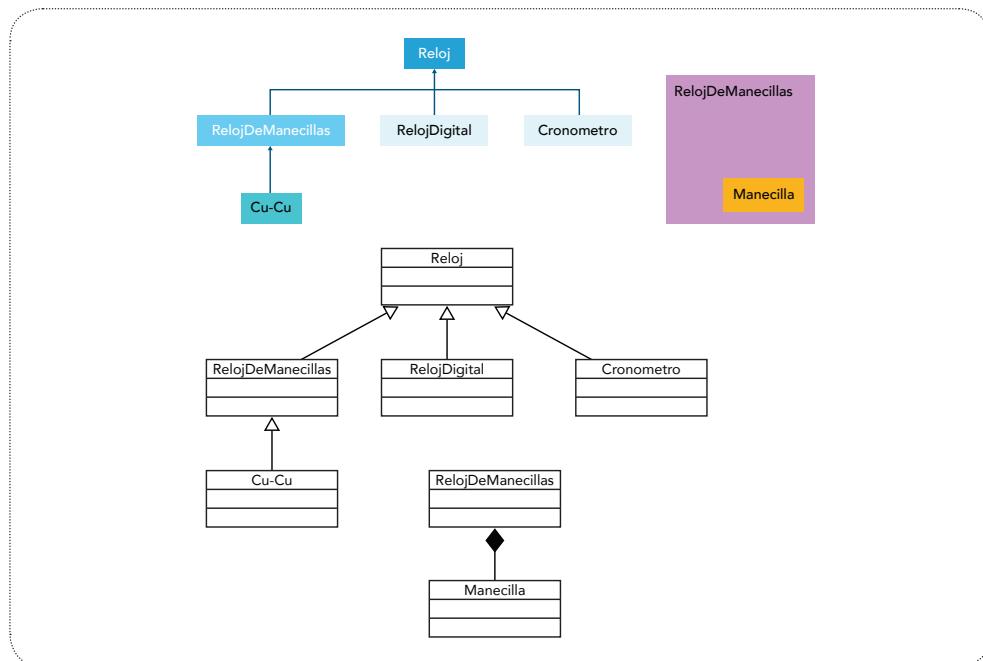


Figura III-17. Relaciones de herencia y agregación del RelojDeManecillas

Recordemos la clase **Manecilla** (sin los atributos largo y ancho ya que no son usados en este ejemplo):

```

public class Manecilla {
    private Integer numero;
    private Integer marca;

    public Manecilla( Integer n, Integer m ) {
        moverAPosicion( n, m );
    }

    public void moverAPosicion ( Integer n, Integer m ) {
        numero = n % 12; // del 0 al 11 y después haremos que el 0
                         // se "vea" como 12
        marca   = m % 5;
    }

    public String toString() {
        Integer n = numero;
        if ( n == 0 ) {
            n = 12; // para que el 0 se "vea" como 12
        }
        return n + "/" + marca;
    }
}
  
```

En la clase `Manecilla` se usan dos enteros: `numero` y `marca`. Estos números indican hacia dónde apunta la manecilla. El `numero` puede estar entre 1 y 12, que son los números visibles en un reloj de manecillas, y la `marca` puede estar entre 0 y 4, que son las marcas que hay entre un número y el siguiente.

En la solución parcial presentamos la estructura de la clase `RelojDeManecillas`.

Solución parcial

```
public class RelojDeManecillas extends Reloj {  
    private Manecilla horario;  
    private Manecilla minutero;  
    private Manecilla segundero;  
  
    RelojDeManecillas( Integer h, Integer m, Integer s ) {  
        super( h, m, s ); //la clase Reloj inicializa los contadores  
  
        // Calcula la posicion de las manecillas respecto a  
        // los numeros en caratula del reloj (0..11)  
        // y a las marcas entre ellos (0..4), donde 0 es en el numero  
        ...  
    }  
  
    public void tic() {  
        super.tic(); // Reloj tiene el funcionamiento de  
                    // los contadores  
  
        // se agrega el comportamiento de las manecillas  
        ...  
    }  
  
    public String toString() {  
        ...  
    }  
}
```

En la solución final vemos que, para determinar la posición de las manecillas, se hace uso (mediante el llamado a `getHora()`, `getMinuto()` y `getSegundo()`) de los contadores cíclicos de la clase padre `Reloj`. Nótese cómo cada manecilla avanza marca por marca.

Solución final

```
public class RelojDeManecillas extends Reloj {  
    private Manecilla horario;  
    private Manecilla minutero;  
    private Manecilla segundero;
```

```
RelojDeManecillas( Integer h, Integer m, Integer s ) {
    super( h, m, s );    //la clase Reloj inicializa los contadores

    // Calcula la posicion de las manecillas respecto a
    // los números en caratula del reloj (0..11)
    // y a las marcas entre ellos (0..4, con 0 es en el número)
    horario = new Manecilla( getHora(), getMinuto() / 12 );
    minutero = new Manecilla( getMinuto() / 5, getMinuto() % 5 );
    segundero = new Manecilla( getSegundo() / 5, getSegundo() % 5 );
}

public void tic() {
    super.tic();    // Reloj tiene el funcionamiento de
                    // los contadores

    // Se agrega el comportamiento de las manecillas
    horario.moverAPosicion( getHora(), getMinuto() / 12 );
    minutero.moverAPosicion( getMinuto() / 5, getMinuto() % 5 );
    segundero.moverAPosicion( getSegundo() / 5,
                                getSegundo() % 5 );
}

public String toString() {
    return "Manec: h:" + horario
        + " m:" + minutero
        + " s:" + segundero;
}
}
```

Ejercicio 3. Implementar la clase Cucu, como descendiente de la clase abstracta Reloj, con las siguientes características:

Tiene, como atributo privado, el objeto `elPajarito` de la clase Pajaro. La clase Pajaro sólo tiene un método que se llama `canta()`, que lo único que hace es poner en pantalla el mensaje “Cu-Cu” para simular, de una manera muy simple, el accionamiento del mecanismo del pajarito que estos relojes tienen. La clase Pajaro queda como sigue:

```
public class Pajaro {
    public void canta() {
        System.out.println( "Cu-Cu " );
    }
}
```

Los métodos de Cucu son:

- El constructor Cucu que, además de crear elPajarito, inicializa los atributos de su padre (RelojDeManecillas).
- El método `tic()` que, además de funcionar como el `tic()` de Reloj, hace que el pajarito cante una vez por segundo cuando el minutero indique la hora “en punto”, es decir, que el minutero esté en la posición “cero” (o 12) y el número de segundos transcurridos sea menor al número de horas de la hora actual. De esta manera, como elPajarito canta una vez cada segundo (tic), lo hará el número de veces que corresponde a la hora en punto (por ejemplo, si son las 7, canta siete veces).

Como observamos en la figura III-18, Cucu tiene todas las propiedades de un Reloj y de un RelojDeManecillas, por lo tanto, también tiene tres manecillas y, con el método `toString()`, convierte la lectura en una cadena de caracteres. Además, tiene un pájaro que canta.

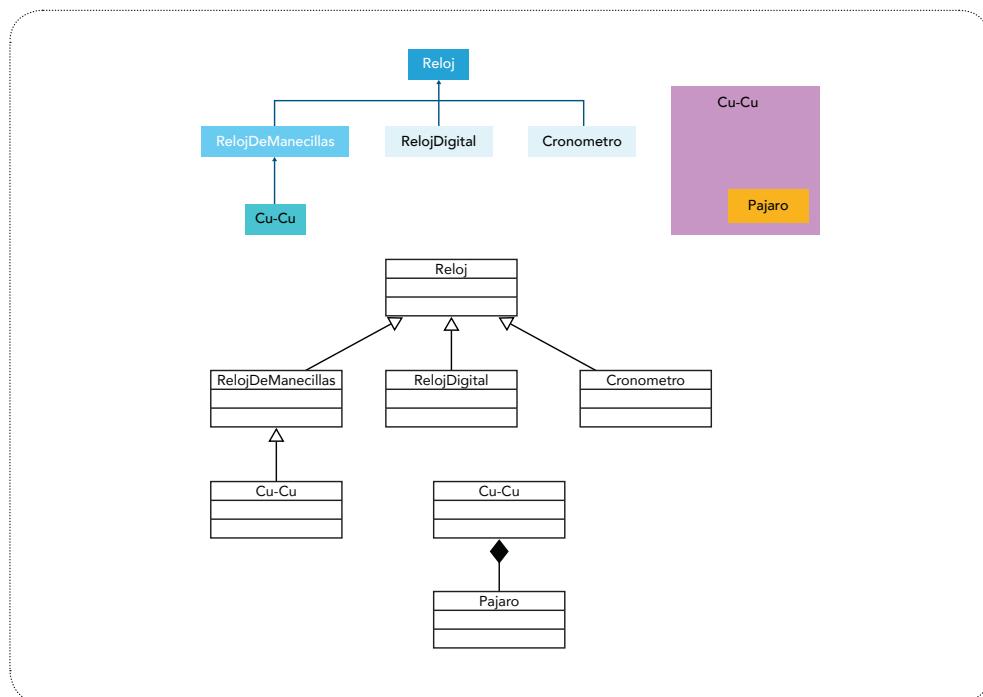


Figura III-18. Relaciones de herencia y agregación del reloj Cu-Cu

En la solución parcial se presenta el esqueleto de la clase Cu-cu.

Solución parcial

```
public class Cucu extends RelojDeManecillas {
    private Pajaro elPajarito;

    public Cucu( Integer h, Integer m, Integer s ) {
        ...
    }

    public void tic() {
        super.tic();
        ...
    }
}
```

Crea elPajarito de clase Pajaro, además de inicializar los atributos de su padre.

elPajarito canta el número de veces que indica la hora cuando el minutero está en la posición cero.

En la solución final, se completan el constructor y el método tic().

Solución final

```
public class Cucu extends RelojDeManecillas {
    private Pajaro elPajarito;

    public Cucu( Integer h, Integer m, Integer s ) {
        super( h, m, s );
        elPajarito = new Pajaro();
    }

    public void tic() {
        super.tic();
        if ( getMinuto() == 0 ) {
            if ( ( getHora() > getSegundo() )
                || ( getHora() == 0 ) && ( getSegundo() < 12 ) ) {
                elPajarito.canta(); // canta una vez
            }
        }
    }
}
```

Ejercicio 4. Implementar la clase RelojDigital como descendiente de la clase abstracta Reloj. RelojDigital tiene como atributo privado un objeto de clase Display.

Como observamos en la figura III-19, RelojDigital tiene todas las propiedades de un Reloj y además tiene un display en el que despliega la hora actual.

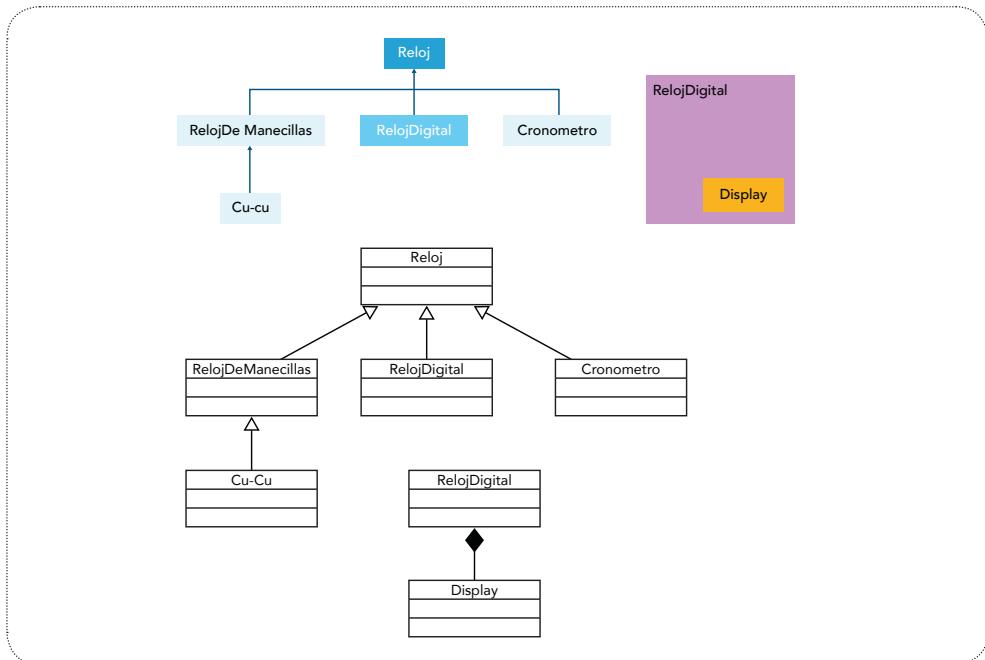


Figura III-19. Relaciones de herencia y agregación del RelojDigital

Tiene los siguientes métodos:

- **RelojDigital()**, que crea **elDisplay** y lo actualiza. También se inicializa como **Reloj**.
- El método **tic()**, que además de funcionar como el **tic()** de **Reloj**, actualiza **elDisplay**.
- **actualizarDisplay()**, que pone en **elDisplay** la hora actual. Además pone la hora en cero cuando son las 12.
- El método **toString()**, que devuelve el contenido de **elDisplay** con sus tres números pegados al indicador de a.m./p.m.

Recordemos la clase **Display**:

```

public class Display {
    private String[] par; // arreglo de pares de dígitos

    public Display() {
        par = new String[3];
        poner( 0, 0, 0 );
    }

    public void poner( Integer h, Integer m, Integer s ) {
        Integer[] num = { h, m, s };
    }
}
  
```

```

for ( int i = 0; i < 3; i++ ) {
    num[i] = num[i] % 100;    // para asegurar un numero valido
    if ( num[i] < 0 ) {
        num[i] = -num[i];
    }
    if ( num[i] < 10 ) {
        par[i] = "0" + num[i].toString();
    }
    else {
        par[i] = num[i].toString();
    }
}
}

public String toString() {
    return par[0] + ":" + par[1] + ":" + par[2];
}
}

```

Completemos la solución parcial que presentamos de la clase RelojDigital.

Solución parcial

```

public class RelojDigital extends Reloj {
    private Display elDisplay;

    RelojDigital( Integer h, Integer m, Integer s ) {
        ...
    }
}

private void actualizarDisplay() {
    Integer h = getHora();
    if( h == 0 ) {
        h = 12;
    }
    Integer[] nums = new Integer[3];
    elDisplay.poner( h, getMinuto( ), getSegundo( ) );
}

public void tic() {
    ...
}

public String toString() {
    String meridiano = "AM";
    if ( ... ) {
        meridiano = "PM";
    }
}

```

Crea elDisplay y lo actualiza.

Funciona como tic() de Reloj pero además actualiza el display.

Usar el método de Reloj que indica si es a.m. o p.m.

```
    }
    return ...
}
}
```

Solución final

```
public class RelojDigital extends Reloj {
    private Display elDisplay;

    RelojDigital( Integer h, Integer m, Integer s ) {
        super( h, m, s );
        elDisplay = new Display();
        actualizarDisplay();
    }

    private void actualizarDisplay() {
        Integer h = getHora();
        if( h == 0 ) {
            h = 12;
        }
        elDisplay.poner( h, getMinuto(), getSegundo() );
    }

    public void tic() {
        super.tic();
        actualizarDisplay();
    }

    public String toString() {
        String meridiano = "AM";
        if( !getAm() ) {
            meridiano = "PM";
        }
        return "Digital: " + elDisplay.toString() + " " + meridiano;
    }
}
```

Ejercicio 5. Implementar la clase **Cronometro** como descendiente de la clase abstracta **Reloj** con los atributos privados siguientes:

- Un objeto de clase **Display**.
- Dos indicadores booleanos, **corriendo** y **congelado**, que indican, respectivamente, verdadero cuando el cronómetro está corriendo y cuando está congelado.

Además tiene los siguientes métodos:

- **Cronometro()**, que además de inicializar los atributos de su ancestro Reloj, crea elDisplay y pone los indicadores corriendo y congelado en falso.
- El método **tic()**, que además de funcionar como el **tic()** de Reloj, actualiza el display.
- **reset()**, que pone el display en ceros.
- **start()**, que pone el indicador **corriendo** en verdadero.
- **stop()**, que pone el indicador **corriendo** en falso.
- **lap()**, que, si el cronómetro está **corriendo** o **congelado**, cambia de estado al indicador **congelado**.
- **tic()**, que, si el indicador **corriendo** es verdadero, hace lo que su padre y si el cronómetro no está **congelado**, entonces pone en el display la cuenta actual.
- **toString()**, que convierte a **String** la lectura con el siguiente formato: "Crono: ##.##:## C L", en donde C está presente sólo si el indicador **corriendo** es verdadero y L sólo si el indicador **congelado** es verdadero.

Como observamos en la figura III-20, Cronometro tiene todas las propiedades de un Reloj y además contienen un Display. Además, tiene métodos que cambian su comportamiento.

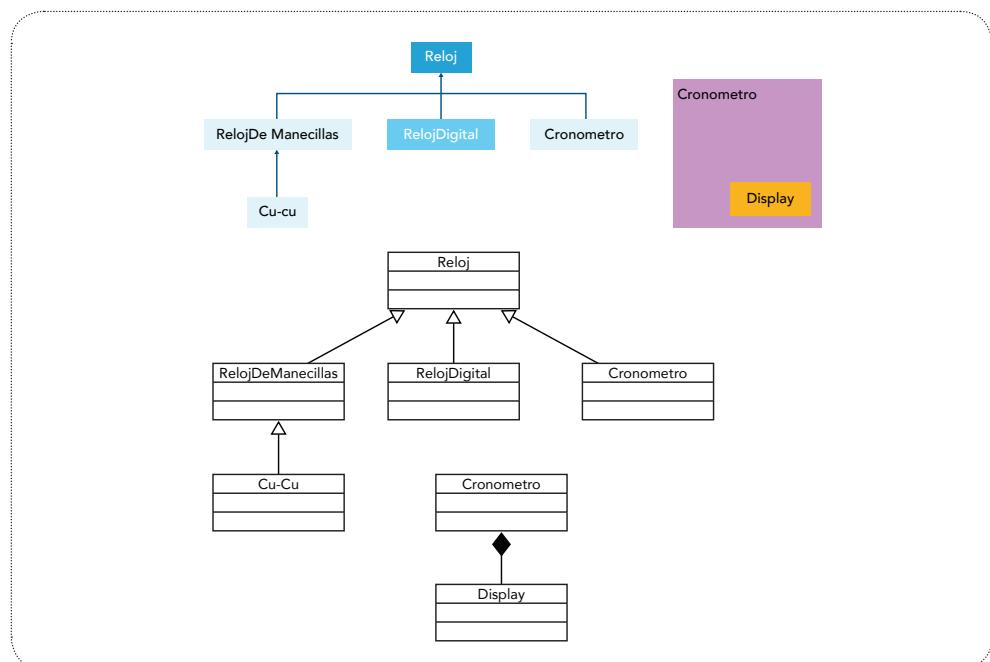


Figura III-20. Relaciones de herencia y agregación del Cronometro

En la primera solución parcial se presenta el esqueleto de la clase **Cronometro**.

Nota: en Java (y también en C/C++) existe el siguiente operador condicional, el cual es una estructura de control *if-else* en forma de operador.

```
<condicion> ? <instruccion1> : <instruccion2>
```

lo anterior es equivalente a:

```
if( condicion )
    instruccion1;
else
    instruccion2;
```

Primera solución parcial:

```
public class Cronometro extends Reloj {
    private Display elDisplay;
    private Boolean corriendo;
    private Boolean congelado;

    public Cronometro() {
        ...
    }                                ← Pone los indicadores en falso
                                         e instancia a elDisplay.

    private void reset() {
        ...
    }                                ← Pone elDisplay en ceros

    public void start() {
        ...
    }                                ← Pone en verdadero el indicador
                                         corriendo

    public void stop() {
        ...
    }                                ← Pone en falso el indicador
                                         corriendo

    public void lap() {
        if ( corriendo || congelado ) {
            ...
        }
    }

    public void tic() {
        if ( corriendo ) {
            ...
        }
    }
}
```

```
        }
        if ( !congelado ) {
            ...
        }
    }

    public String toString() {
        return ...
    }
}
```

Segunda solución parcial:

```
public class Cronometro extends Reloj {
    private Display elDisplay;
    private Boolean corriendo;
    private Boolean congelado;

    public Cronometro() {
        super( 0, 0, 0 );
        corriendo = false;
        congelado = false;
        elDisplay = new Display();
    }

    private void reset() {
        ponerALas( 0, 0, 0 );
    }

    public void start() {
        corriendo = true;
    }

    public void stop() {
        corriendo = false;
    }

    public void lap() {
        if( corriendo || congelado ) {
            ...
        }
    }

    public void tic() {
        if( corriendo ) {
            ...
        }
        if( !congelado ) {
            ...
        }
    }
}
```

```
    }
}

public String toString() {
    return ...
}
}
```

Completemos la segunda solución parcial de acuerdo con los requerimientos.

Solución final

```
public class Cronometro extends Reloj {
    private Display elDisplay;
    private Boolean corriendo;
    private Boolean congelado;

    public Cronometro() {
        super( 0, 0, 0 );
        corriendo = false;
        congelado = false;
        elDisplay = new Display();
    }

    private void reset() {
        ponerALas( 0, 0, 0 );
    }

    public void start() {
        corriendo = true;
    }

    public void stop() {
        corriendo = false;
    }

    public void lap() {
        if ( corriendo || congelado ) {
            congelado = !congelado;
        } else {
            reset();
        }
    }

    public void tic() {
        if ( corriendo ) {
            super.tic();
        }
    }
}
```

```
    if ( !congelado ) {
        elDisplay.poner( getHora(), getMinuto(), getSegundo() );
    }
}

public String toString() {
    return "Crono: " + elDisplay.toString()
        + ( corriendo ? " C" : " " )
        + ( congelado ? " L" : " " );
}
}
```

Ahora escribimos la clase principal `RelojesMain` para crear diferentes tipos de `Reloj` y llamar algunos de sus métodos:

```
public class RelojesMain {

    // Este metodo llama n veces al metodo tic de cada reloj que
    // recibe como parametro
    private static void tic( Cucu reloj1, Cronometro reloj2,
                            RelojDigital reloj3, Integer n ) {
        for ( int i = 0; i < n; i++ ) {
            reloj1.tic();
            reloj2.tic();
            reloj3.tic();
            System.out.println( reloj1 + "\t\t\t"
                + reloj2 + "\t\t\t"
                + reloj3 );
        }
    }

    public static void main( String[] args ) {
        // Hace varias pruebas con tres objetos derivados de Reloj
        Cucu reloj1 = new Cucu( 2, 59, 59 );
        Cronometro reloj2 = new Cronometro();
        RelojDigital reloj3 = new RelojDigital( 11, 58, 57 );
        tic( reloj1, reloj2, reloj3, 5 );
        reloj2.start();
        tic( reloj1, reloj2, reloj3, 5 );
        reloj2.lap();
        tic( reloj1, reloj2, reloj3, 5 );
        reloj2.lap();
        tic( reloj1, reloj2, reloj3, 5 );
        reloj2.stop();
        tic( reloj1, reloj2, reloj3, 5 );
        reloj2.start();
        tic( reloj1, reloj2, reloj3, 5 );
        reloj2.lap();
        tic( reloj1, reloj2, reloj3, 5 );
    }
}
```

```

reloj2.stop();
tic(reloj1, reloj2, reloj3, 5 );
reloj2.lap();
tic(reloj1, reloj2, reloj3, 5 );
reloj2.start();
tic(reloj1, reloj2, reloj3, 5 );
reloj2.stop();
tic(reloj1, reloj2, reloj3, 5 );
reloj2.lap();
tic(reloj1, reloj2, reloj3, 5 );
}
}

```

Ejercicio 6. Determina, sin ver la solución, cuál será la salida del programa anterior.

Solución

Cu-Cu		
Manec: h:3/0 m:12/0 s:12/0	Crono: 00:00:00	Digital: 11:58:58 AM
Cu-Cu		
Manec: h:3/0 m:12/0 s:12/1	Crono: 00:00:00	Digital: 11:58:59 AM
Cu-Cu		
Manec: h:3/0 m:12/0 s:12/2	Crono: 00:00:00	Digital: 11:59:00 AM
Manec: h:3/0 m:12/0 s:12/3	Crono: 00:00:00	Digital: 11:59:01 AM
Manec: h:3/0 m:12/0 s:12/4	Crono: 00:00:00	Digital: 11:59:02 AM
Manec: h:3/0 m:12/0 s:1/0	Crono: 00:00:01 C	Digital: 11:59:03 AM
Manec: h:3/0 m:12/0 s:1/1	Crono: 00:00:02 C	Digital: 11:59:04 AM
Manec: h:3/0 m:12/0 s:1/2	Crono: 00:00:03 C	Digital: 11:59:05 AM
Manec: h:3/0 m:12/0 s:1/3	Crono: 00:00:04 C	Digital: 11:59:06 AM
Manec: h:3/0 m:12/0 s:1/4	Crono: 00:00:05 C	Digital: 11:59:07 AM
Manec: h:3/0 m:12/0 s:2/0	Crono: 00:00:05 C L	Digital: 11:59:08 AM
Manec: h:3/0 m:12/0 s:2/1	Crono: 00:00:05 C L	Digital: 11:59:09 AM
Manec: h:3/0 m:12/0 s:2/2	Crono: 00:00:05 C L	Digital: 11:59:10 AM
Manec: h:3/0 m:12/0 s:2/3	Crono: 00:00:05 C L	Digital: 11:59:11 AM
Manec: h:3/0 m:12/0 s:2/4	Crono: 00:00:05 C L	Digital: 11:59:12 AM
Manec: h:3/0 m:12/0 s:3/0	Crono: 00:00:11 C	Digital: 11:59:13 AM
Manec: h:3/0 m:12/0 s:3/1	Crono: 00:00:12 C	Digital: 11:59:14 AM
Manec: h:3/0 m:12/0 s:3/2	Crono: 00:00:13 C	Digital: 11:59:15 AM
Manec: h:3/0 m:12/0 s:3/3	Crono: 00:00:14 C	Digital: 11:59:16 AM
Manec: h:3/0 m:12/0 s:3/4	Crono: 00:00:15 C	Digital: 11:59:17 AM
Manec: h:3/0 m:12/0 s:4/0	Crono: 00:00:15	Digital: 11:59:18 AM
Manec: h:3/0 m:12/0 s:4/1	Crono: 00:00:15	Digital: 11:59:19 AM
Manec: h:3/0 m:12/0 s:4/2	Crono: 00:00:15	Digital: 11:59:20 AM
Manec: h:3/0 m:12/0 s:4/3	Crono: 00:00:15	Digital: 11:59:21 AM
Manec: h:3/0 m:12/0 s:4/4	Crono: 00:00:15	Digital: 11:59:22 AM
Manec: h:3/0 m:12/0 s:5/0	Crono: 00:00:16 C	Digital: 11:59:23 AM
Manec: h:3/0 m:12/0 s:5/1	Crono: 00:00:17 C	Digital: 11:59:24 AM
Manec: h:3/0 m:12/0 s:5/2	Crono: 00:00:18 C	Digital: 11:59:25 AM
Manec: h:3/0 m:12/0 s:5/3	Crono: 00:00:19 C	Digital: 11:59:26 AM
Manec: h:3/0 m:12/0 s:5/4	Crono: 00:00:20 C	Digital: 11:59:27 AM

```

Manec: h:3/0 m:12/0 s:6/0 Crono: 00:00:20 C L Digital: 11:59:28 AM
Manec: h:3/0 m:12/0 s:6/1 Crono: 00:00:20 C L Digital: 11:59:29 AM
Manec: h:3/0 m:12/0 s:6/2 Crono: 00:00:20 C L Digital: 11:59:30 AM
Manec: h:3/0 m:12/0 s:6/3 Crono: 00:00:20 C L Digital: 11:59:31 AM
Manec: h:3/0 m:12/0 s:6/4 Crono: 00:00:20 C L Digital: 11:59:32 AM
Manec: h:3/0 m:12/0 s:7/0 Crono: 00:00:20 L Digital: 11:59:33 AM
Manec: h:3/0 m:12/0 s:7/1 Crono: 00:00:20 L Digital: 11:59:34 AM
Manec: h:3/0 m:12/0 s:7/2 Crono: 00:00:20 L Digital: 11:59:35 AM
Manec: h:3/0 m:12/0 s:7/3 Crono: 00:00:20 L Digital: 11:59:36 AM
Manec: h:3/0 m:12/0 s:7/4 Crono: 00:00:20 L Digital: 11:59:37 AM
Manec: h:3/0 m:12/0 s:8/0 Crono: 00:00:25 Digital: 11:59:38 AM
Manec: h:3/0 m:12/0 s:8/1 Crono: 00:00:25 Digital: 11:59:39 AM
Manec: h:3/0 m:12/0 s:8/2 Crono: 00:00:25 Digital: 11:59:40 AM
Manec: h:3/0 m:12/0 s:8/3 Crono: 00:00:25 Digital: 11:59:41 AM
Manec: h:3/0 m:12/0 s:8/4 Crono: 00:00:25 Digital: 11:59:42 AM
Manec: h:3/0 m:12/0 s:9/0 Crono: 00:00:26 C Digital: 11:59:43 AM
Manec: h:3/0 m:12/0 s:9/1 Crono: 00:00:27 C Digital: 11:59:44 AM
Manec: h:3/0 m:12/0 s:9/2 Crono: 00:00:28 C Digital: 11:59:45 AM
Manec: h:3/0 m:12/0 s:9/3 Crono: 00:00:29 C Digital: 11:59:46 AM
Manec: h:3/0 m:12/0 s:9/4 Crono: 00:00:30 C Digital: 11:59:47 AM
Manec: h:3/0 m:12/0 s:10/0 Crono: 00:00:30 Digital: 11:59:48 AM
Manec: h:3/0 m:12/0 s:10/1 Crono: 00:00:30 Digital: 11:59:49 AM
Manec: h:3/0 m:12/0 s:10/2 Crono: 00:00:30 Digital: 11:59:50 AM
Manec: h:3/0 m:12/0 s:10/3 Crono: 00:00:30 Digital: 11:59:51 AM
Manec: h:3/0 m:12/0 s:10/4 Crono: 00:00:30 Digital: 11:59:52 AM
Manec: h:3/0 m:12/0 s:11/0 Crono: 00:00:00 Digital: 11:59:53 AM
Manec: h:3/0 m:12/0 s:11/1 Crono: 00:00:00 Digital: 11:59:54 AM
Manec: h:3/0 m:12/0 s:11/2 Crono: 00:00:00 Digital: 11:59:55 AM
Manec: h:3/0 m:12/0 s:11/3 Crono: 00:00:00 Digital: 11:59:56 AM
Manec: h:3/0 m:12/0 s:11/4 Crono: 00:00:00 Digital: 11:59:57 AM

```

Prácticas de laboratorio

En esta sección se retomarán las dos prácticas presentadas en el capítulo anterior, pero incluyendo los temas estudiados en éste.

Robot

Redefinición de la clase Robot

Originalmente, la posición del *Robot* está dada por dos atributos, *x* y *y*, de tipo *Integer*. Ahora, para modelar de mejor forma, y eventualmente poder reutilizar código, es necesaria una clase *Punto*, cuyos atributos son dos números *Float* que representan las coordenadas (*x*, *y*) de un punto en un espacio bidimensional continuo. Esta clase, además de contener sus respectivos métodos *getters* y *setters*, considera el método:

- `esIgual(Punto p)`, que devuelve `true` si las coordenadas (`x, y`) de un objeto `Punto` son iguales a las del punto `p` que recibe como parámetro y `false` en caso contrario.

De esta forma, el `Robot` ya no tiene dos atributos `Integer` que representan su posición, sino solamente uno de tipo `Punto`. El `Robot` sigue considerando, adicionalmente a los métodos `getters` y `setters`, los métodos `avanzar()`, para hacerlo avanzar una posición en su dirección actual, y `girar()`, para hacerlo girar 90 grados en sentido contrario a las manecillas del reloj.

Además de los constructores originales, con la redefinición del atributo `posición`, se tienen que considerar el siguiente:

- `Robot(Punto pos, int dir)`, constructor que establece los valores en los parámetros `pos` y `dir` para los atributos `posicion` y `direccion`, respectivamente.

Implementa la clase `Punto` y la redefinición de la clase `Robot` considerando los atributos, constructores y métodos descritos.

Especialización de la clase Robot

Consideremos ahora el escenario que se muestra en la figura III-21. Este escenario consiste en lo siguiente:

- Un espacio bidimensional discreto de $2m + 1$ filas y $2n + 1$ columnas, que contiene algunos obstáculos.
- Un robot ubicado en alguna posición del espacio bidimensional y que tiene que desplazarse hasta una posición objetivo.

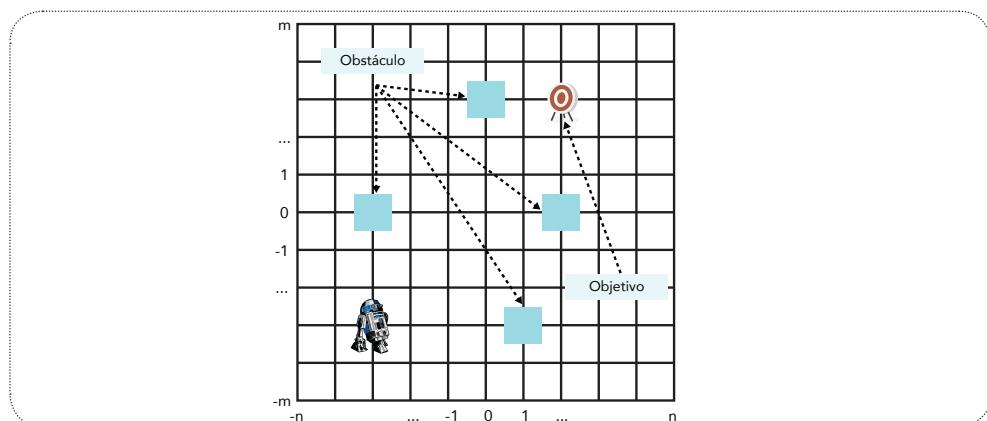


Figura III-21. Escenario con un espacio bidimensional y un robot

Para modelar este escenario, necesitamos considerar las siguientes clases:

- **Espacio**, que tiene como atributos las dimensiones *m* y *n*, y las posiciones de los *obstáculos* que están presentes (tal vez como un arreglo suficientemente grande de *Posiciones*).
- **SuperRobot**, que tiene como atributos su *posicion* y su *direccion* (similar al Robot redefinido antes).
-

Las clases **Espacio** y **SuperRobot** tienen una relación de asociación, porque **SuperRobot** *usa* un **Espacio** para llegar a su posición objetivo y **Espacio** *usa* a un **SuperRobot** para saber si una posición está libre. Sin embargo, un **Espacio** no es un atributo de un **SuperRobot** y un **SuperRobot** no es un atributo de un **Espacio**.

Por lo tanto, además de los *getters* y *setters*, la clase *Espacio* considera los siguientes métodos:

- **Espacio(m, n)**, método constructor que crea un espacio bidimensional de $2m + 1$ filas y $2n + 1$ columnas.
- **agregarObstaculo(Posicion pos)**, agrega un obstáculo en la posición *pos*. Devuelve *true* si se pudo agregar el obstáculo y *false* si la posición *pos* está ocupada, ya sea por un obstáculo o por un robot.
- **estaLibre(Posicion pos)**, verifica si la posición *pos* está libre de obstáculos y de robots, en cuyo caso devuelve *true* y, en caso contrario, devuelve *false*.
- **asociar(SuperRobot r)**, asocia al **SuperRobot** *r* con este **Espacio** para representar la relación de asociación.

La clase **SuperRobot** puede verse como una especialización de la clase **Robot**, pues tiene los métodos *getters* y *setters*, **girar()** y **avanzar()** definidos en ésta. Adicionalmente, considera los siguientes métodos:

- **girarAlReves()**, que hace girar al **Robot** 90 grados en el sentido de las manecillas del reloj.
- **saltar()**, con el cual el **Robot** avanza, en un solo movimiento, dos posiciones en su dirección actual.
- **asociar(Espacio e)**, para representar la relación de asociación de este **SuperRobot** con el **Espacio** *e*.

Implementa las clases **Espacio** y **SuperRobot** considerando los atributos, constructores y métodos descritos. Además, escribe una clase principal que considere un

Espacio con algunos obstáculos y un SuperRobot que se desplace desde una posición origen a una posición objetivo en dicho espacio.

Tienda virtual

Redefinición de las clases Libro y Pelicula

En el capítulo anterior, el atributo autor de Libro y los atributos protagonista y director de Película se definieron como de tipo String. Dado que estos atributos contienen el nombre de una persona, esta definición puede no ser homogénea. Por lo tanto, un mejor modelo sería considerar la clase adicional:

- Persona, cuyos atributos son nombre y apellido, de tipo String.

Esta clase, además de contener sus respectivos métodos *getters* y *setters*, considera el método:

- `esIgual(Persona p)`, que devuelve `true` si el nombre y apellido de un objeto Persona son iguales a las de la persona p y `false` en caso contrario.

De esta forma, las clases Libro y Pelicula se tienen que redefinir como sigue:

- Libro, cuyos atributos son autor de tipo Persona, titulo de tipo String y precio de tipo Float.
- Pelicula, con los atributos protagonista y director de tipo Persona, titulo de tipo String y precio de tipo Float.

Implementa la clase Persona y la redefinición de las clases Libro y Pelicula, considerando los atributos y métodos descritos.

Generalización de las clases Libro y Pelicula

Como nos podemos dar cuenta, las clases Libro y Pelicula tienen dos atributos en común: `titulo` y `precio`. Entonces, podemos considerar una generalización de estas dos clases para crear la superclase:

- Producto, cuyos atributos son el `titulo` y el `precio` de un producto, de tipo String y Float, respectivamente. Generalmente los productos están relacionados con un identificador único, por lo tanto esta clase también tiene el atributo `id` de tipo Integer.

De esta forma, ahora las clases **Libro** y **Pelicula** son subclases de **Producto** y deben heredar de ésta sus atributos y métodos.

Además, dado que los productos se van a vender, es necesario contar con un catálogo que los clientes puedan revisar. Por esta razón, se considera la clase:

- **Catalogo**, que cuenta con un atributo **productos**, que contiene a los libros y películas en venta (por ejemplo, un arreglo suficientemente grande de **Productos**), y con otro para conocer la cantidad de productos que están disponibles.

Todas las clases anteriores consideran sus respectivos constructores, métodos *getters* y *setters* y el método **toString()**. Adicionalmente, la clase **Catalogo** considera los siguientes métodos:

- **agregar(Producto p)**, agrega el **Producto** *p* al catálogo.
- **eliminar(Integer id)**, elimina el producto cuyo identificador único es *id*.
- **buscar(String titulo)**, devuelve, contenidos en un **Catalogo**, a todos los **Productos** cuyo título es *titulo*.
- **buscar(Persona p)**, devuelve, contenidos en un **Catalogo**, a todos los **Productos** cuyo autor, director o protagonista, según sea el caso, es *p*.

Implementa las clases **Persona**, **Libro** y **Catalogo**, considerando los atributos, constructores y métodos descritos.

Cuestionario

Contesta las siguientes preguntas:

1. ¿En qué consiste la herencia?
2. ¿Qué son las clases abstractas y para qué se usan?
3. Supongamos que un objeto de clase A está agregado a un objeto de clase B, ¿si borramos el objeto de clase B, desaparece también el objeto de clase A?
4. Supongamos que un objeto de clase B está compuesto por varios objetos de clase A, ¿si borramos el objeto de clase B, desaparecen también los objetos de clase A?

Uso de algunas clases predefinidas en Java

Objetivos

Reutilizar clases definidas en librerías de Java para:

- Capturar datos del teclado.
- Leer y escribir en archivos de texto.
- Trabajar con arreglos.
- Trabajar con listas ligadas, pilas y colas.

Introducción

Una de las ventajas del lenguaje Java es que ya tiene definidas una gran cantidad de clases para su reutilización. Estas clases predefinidas están organizadas por bibliotecas. En este capítulo estudiamos sólo las clases útiles para un curso introductorio de POO, que son las que son útiles para capturar datos del teclado, trabajar con archivos, arreglos, listas ligadas, pilas y colas.

Excepciones

La entrada y salida de datos es una de las características principales de cualquier sistema de cómputo. Estudiaremos algunas de las clases ya definidas en Java, que son las que sirven para capturar datos del teclado y para trabajar con archivos, tanto de texto como binarios. Antes de comenzar, es necesario estudiar un tema que es un requisito para la entrada y salida de datos, que es el de las excepciones.

El término *excepción* se entiende como un *evento no esperado*, el cual se produce durante la ejecución de un programa y que interrumpe el flujo normal de las instrucciones. Cuando se produce una excepción dentro de un método, éste crea un objeto y se lo entrega al sistema de ejecución. Este objeto, llamado un *objeto excepción*, contiene información sobre la excepción, incluyendo su tipo y el estado del programa cuando éste se produjo. A la creación de un objeto excepción y la entrega al sistema de ejecución se le llama *arrojar una excepción*. Después de que un método arroja una excepción, el sistema de ejecución de programas busca en la *pila de invocaciones a métodos* un método que contenga un bloque de código que pueda "manejar" la excepción. Este bloque de código se llama *manejador de excepciones*.

La búsqueda comienza en el método en el que se produjo la excepción y procede a través de la *pila de invocaciones a métodos* en el orden inverso en el que los métodos fueron invocados. Cuando se encuentra un manejador de excepciones adecuado, el sistema pasa la excepción a éste y le cede el control de la ejecución del programa. Un manejador de excepciones se considera adecuado si el tipo del objeto de la excepción generada coincide con el tipo que puede manejar.

El manejador de excepciones seleccionado se dice que *atrapa la excepción*. Si el sistema de ejecución busca exhaustivamente en todos los métodos de la pila de invocaciones, sin encontrar un manejador de excepciones apropiado, el sistema de ejecución, y en consecuencia el programa, termina. Es decir, con las excepciones se “intenta” la ejecución de un bloque y, en caso de que se presente algún caso excepcional, se “arroja” una señal de excepción en un objeto que deberá ser “atrapada” fuera del bloque mencionado. Así, el manejo de excepciones se da utilizando los bloques `try`, `catch` y `finally`. El bloque `try` incluye las líneas de código que son susceptibles a situaciones excepcionales, es decir, en donde pueda ocurrir una excepción cuando algo anormal sucede. El siguiente código muestra un ejemplo de un bloque `try`, dentro del cual se intenta abrir un archivo:

```
fr = null;
try {
    ...
    FileReader fr = new FileReader("archivo.txt");
    ...
}
catch (FileNotFoundException e) {
    ...
}
finally {
    ...
}
```

Se intentará abrir `archivo.txt`. con el constructor de `FileReader`. Si no se logra abrir, el constructor arrojará una excepción de tipo `FileNotFoundException`.

El bloque `catch` atrapa la excepción (`FileNotFoundException` en nuestro ejemplo) que fue arrojada dentro del bloque `try` y se le da el tratamiento adecuado:

```
fr = null;
try {
    ...
    FileReader fr = new FileReader("archivo.txt");
    ...
}
catch (FileNotFoundException e) {
    System.err.println("Error: " + e.getMessage());
}
```

La excepción se atrapa en el objeto `e`.

Enviamos el mensaje de error que está en el objeto `e`.

```

    }
    finally {
        ...
    }
}

```

Además, existe el bloque opcional `finally`, el cual sirve para que se ejecute un fragmento de código independientemente de si se produce una excepción o no. Un ejemplo de esto sería cerrar el archivo con el que estamos trabajando (porque nunca debe quedarse abierto), como se muestra a continuación:

```

fr = null;
try {
    ...
    // Se presenta una excepcion porque el archivo
    // indicado por alguna razón no se puede abrir
    FileReader fr = new FileReader("archivo.txt");
    ...
}
catch (FileNotFoundException e) {

    System.err.println("Error: " + e.getMessage());

}
finally {
    if (fr != null) {
        fr.close();
    }
}

```

Como `fr` se inicializó en `null`, si ya no tiene ese valor, entonces se pudo y hay que cerrar el archivo.

La sintaxis completa para el manejo de excepciones es la siguiente:

```

try {
    // Líneas en donde podria aparecer una excepcion
}
catch( TipoExcepcionA e ) {
    // Manejo del TipoExcepcion A
}
catch( TipoExcepcionB e ) {
    // Manejo del TipoExcepcion B
}
finally {
    // Liberar recursos
    // Este bloque siempre se ejecuta
}

```

Aunque es posible agregar tantos bloques `catch` (asociados al mismo bloque `try`) como se deseen, no hay que abusar de su uso, ya que pierde claridad el código. *Las excepciones se utilizan sólo para controlar casos de excepción del sistema, por lo que hay que evitar usarlas para controlar el flujo del programa.*

Entrada y salida de datos como texto (teclado y pantalla)

Existen tres dispositivos de entrada/salida que se clasifican de acuerdo con su función:

- *Standard input.* Dispositivo de entrada de datos estándar (por defecto es el teclado). `System.in` que es de tipo `InputStream`.
- *Standard output.* Dispositivo a donde se envían los datos de salida (la salida predeterminada es la pantalla). `System.out` que es de tipo `OutputStream`.
- *Standard error.* Dispositivo a donde se envían los mensajes de error (por defecto es la pantalla). `System.err` que es de tipo `OutputStream`.

Un *stream* es un *torrente* o flujo de datos. Esto significa que existe un flujo de datos del teclado hacia el CPU, en el caso de los datos de entrada, y del CPU a la pantalla, en el caso de los datos de salida y de los mensajes de error.

El paquete `java.io` contiene una colección de clases que permiten leer de *streams* de entrada y escribir en *streams* de salida. En esta Sección usaremos las siguientes:

- La clase `InputStreamReader`, o *lector de stream de entrada*, define objetos que leen los bytes de un *stream* y los interpreta como caracteres.
- La clase `BufferedReader`, o *lector acumulador*, hace uso de un objeto de la clase `InputStreamReader` y es capaz de acumular caracteres y, así, de leer cadenas de caracteres (`String`). El método `readLine()` de esta clase regresa un `String` que contiene la siguiente línea de texto leída del *stream*.

Lectura de datos del teclado mediante `InputStreamReader`

Para capturar un dato, declaramos un objeto de la clase `InputStreamReader` asociándolo al teclado (`System.in`). Luego creamos un objeto de la clase `BufferedReader`, asociándolo al anterior `InputStreamReader` y, con este nuevo objeto, hacemos lecturas de líneas completas, como se muestra a continuación:

```
// Se define un flujo de caracteres de entrada
InputStreamReader isr = new InputStreamReader( System.in );
BufferedReader reader = new BufferedReader( isr );

// Se lee la entrada y finaliza al pulsar la tecla Enter
String cadena = reader.readLine();
```

Para leer una cadena de caracteres definimos el método `capturarLinea()` en la clase `LectorDeTeclado`, el cual regresa una cadena con los caracteres capturados del teclado:

```
public class LectorDeTeclado {
    BufferedReader reader;

    public LectorDeTeclado() {
        InputStreamReader isr = new InputStreamReader( System.in );
        reader = new BufferedReader( isr );
    }

    public String capturarLinea() {
        // Aquí vamos a agregar más cosas
        ...

        // La captura finaliza al pulsar la tecla Enter
        return reader.readLine();
    }
}
```

Para capturar datos numéricos (`Integer`, `Double`, etc.) se recomienda *manejar excepciones*, esto con el fin de detectar cuando los datos de entrada no correspondan a un dato numérico y no se puede identificar de qué número se trata.

Ahora construyamos el método `capturarEntero()`, para leer números enteros desde el teclado, dentro de la clase `LectorDeTeclado`:

```
public class LectorDeTeclado {
    BufferedReader reader;

    public LectorDeTeclado() {
        InputStreamReader isr = new InputStreamReader( System.in );
        reader = new BufferedReader( isr );
    }

    public String capturarLinea() {
        // Aquí vamos a agregar más cosas
        ...

        // La captura finaliza al pulsar la tecla Enter
        return reader.readLine();
```

```

}

// Método para capturar enteros
public Integer capturarEntero() {
    try {
        return Integer.parseInt( capturarLinea() );
    }
    catch ( NumberFormatException e ) {
        System.err.println( "Error 1: " + e.getMessage() );
        return Integer.MIN_VALUE; // valor entero más pequeño
    }
}

```

Allamar al método `capturarLinea()` se obtiene de regreso un `String`.

El método `parseInt()` convierte un `String` a su valor correspondiente tipo `Integer`.

En caso de que la cadena de caracteres leída no represente a un número, el método `parseInt()` lanza una excepción de tipo `NumberFormatException` y, en el `catch` de ese tipo de excepción, lo identificamos como "Error 1" en el mensaje al usuario. En este caso, el método `capturarEntero()` regresa un número que está definido como `Integer.MIN_VALUE` y que representa el valor entero más pequeño que se puede usar.

También agregaremos a la clase `LectorDeTeclado` el siguiente método para leer números de punto flotante de precisión doble (`Double`):

```

// Método para capturar dobles
public Double capturarDoble() {
    try {
        return Double.parseDouble( capturarLinea() );
    }
    catch ( NumberFormatException e ) {
        System.err.println( "Error 3: " + e.getMessage() );
        return Double.NaN; // Indica que no es un número;
    }
}

```

El método `parseDouble()` convierte la cadena a su valor de tipo real.

Cuando se atrapa la excepción, se le identifica como "Error 3" y se regresa un valor que está definido como `Double.NaN`, el cual indica que no se trata de un número. A continuación presentamos la clase `LectorDeTeclado` completa.

```

import java.io.*;

public class LectorDeTeclado {
    BufferedReader reader;

    LectorDeTeclado() {
        InputStreamReader isr = new InputStreamReader( System.in );
        reader = new BufferedReader( isr );
    }
}

```

```
}

public String capturarLinea() {
    try {
        // La captura finaliza al pulsar Enter
        return reader.readLine();
    }
    catch ( IOException e ) {
        System.err.println( "Error: 0" + e.getMessage() );
        return null;
    }
}

// Método para capturar enteros
public Integer capturarEntero() {
    try {
        return Integer.parseInt( capturarLinea() );
    }
    catch ( NumberFormatException e ) {
        System.err.println( "Error 1: " + e.getMessage() );
        return Integer.MIN_VALUE;    // valor mas pequeño
    }
}

// Método para capturar flotantes
public Float capturarFlotante() {
    try {
        return Float.parseFloat( capturarLinea() );
    }
    catch ( NumberFormatException e ) {
        System.err.println( "Error 2: " + e.getMessage() );
        return Float.NaN;    // Indica no es un numero
    }
}

// Metodo para capturar dobles
public double capturarDoble() {
    try {
        return Double.parseDouble( capturarLinea() );
    }
    catch ( NumberFormatException e ) {
        System.err.println( "Error 3: " + e.getMessage() );
        return Double.NaN;    // Indica no es un numero
    }
}
```

La siguiente clase principal es un ejemplo de cómo se utiliza la clase `LectorDeTeclado`:

```
public class LeerMain {  
    private static LectorDeTeclado input;  
  
    public static void main( String[] args ) {  
        input = new LectorDeTeclado();  
        Integer i;  
        Double r;  
        i = input.capturarEntero();  
        r = input.capturarDoble();  
    }  
}
```

Lectura de datos del teclado mediante Scanner

La clase **Scanner** define un escáner de texto simple, que puede analizar tipos de datos primitivos y cadenas de caracteres utilizando expresiones regulares. Un **Scanner** divide su entrada en *tokens* utilizando un patrón delimitador, que por defecto coincide con el espacio en blanco. Los *tokens* resultantes pueden entonces ser convertidos en valores de diferentes tipos utilizando los diversos métodos **next**.

En esta sección estudiaremos el uso de la clase **Scanner** cuando los datos provienen del teclado. Para poder utilizar esta clase debemos importarla de la siguiente librería:

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

Al crear un objeto de clase **Scanner** es necesario indicar que los datos provienen del teclado enviando el parámetro **System.in** al constructor.

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

Los métodos más importantes de la clase **Scanner** son:

- **nextLine()**, que lee y devuelve un **String**.
- **nextInt()**, que lee y devuelve un **int**.
- **nextDouble()**, que lee un y devuelve un **double**.
- **nextFloat()**, que lee y devuelve un **float**.

Cabe señalar que con **Scanner** no se pueden leer *wrappers*. A continuación presentamos un programa de ejemplo con el uso de la clase **Scanner**.

```
import java.util.Scanner;  
  
public class ScannerMain {
```

```
public static void main( String[] args ) {  
  
    Scanner input = new Scanner( System.in );  
    String nombre;  
    double radio;  
    int n;  
    float flotante ;  
  
    System.out.print( "¿Como te llamas? " );  
    nombre = input.nextLine(); // lee un String  
    System.out.println( "¿Como estas, " + nombre + "?" );  
  
    System.out.print( "Cual es el radio del circulo? " );  
    radio = input.nextDouble(); // lee un double  
    System.out.println( "La longitud de la circunferencia es: "  
        + 2 * Math.PI * radio );  
  
    System.out.print( "Ahora dame un numero entero: " );  
    n = input.nextInt(); // lee un entero  
    System.out.println( "El cuadrado del entero es: "  
        + ( n * n ) );  
  
    System.out.println( "Ahora dame un numero de punto "  
        + "flotante:" );  
    flotante = input.nextFloat();  
    System.out.println( "Tu numero es: " + flotante );  
}  
}
```

Ejercicio 1. Determina cuál será la salida del programa anterior, sin ver la solución, cuando los datos tecleados sean "Mario", 5, 76 y 45.72.

Solución:

A continuación se muestra un ejemplo de salida de este programa.

```
¿Como te llamas? Mario  
¿Como estas, Mario?  
Cual es el radio del circulo? 5  
La longitud de la circunferencia es: 31.41592653589793  
Ahora dame un numero entero: 76  
El cuadrado del entero es: 5776.0  
Ahora dame un número de punto flotante:  
45.72  
Tu número (float) es: 45.72
```

Los métodos `nextDouble()`, `nextInt()` y `nextFloat()` usados en el programa anterior, podrían arrojar tres tipos de excepciones:

- `InputMismatchException`, si el siguiente token no coincide con la expresión regular correspondiente a cada tipo de dato o si el dato está fuera de rango.
- `NoSuchElementException`, si no hay más *tokens* en la entrada.
- `IllegalStateException`, si este `Scanner` está cerrado.

El método `nextLine()` sólo arroja los últimos dos tipos de excepciones. Por lo tanto, para un manejo adecuado de excepciones, el programa anterior debería considerar la inclusión de los bloques `try`, `catch` y `finally`:

```
import java.util.Scanner;
import java.io.*;

public class ArchivosMain {

    public static void main( String[] args ) {
        PrintWriter pw = null;

        try {
            File archivoEntrada = new File( "datos.txt" );
            FileWriter archivoSalida = new FileWriter( "salida.txt" );
            Scanner scanner = new Scanner( archivoEntrada );
            pw = new PrintWriter( archivoSalida );
            int cont = 0;

            while ( scanner.hasNextLine() ) {
                // Leemos el dato del archivo
                int dato = scanner.nextInt();
                cont++;
                // Guardamos el dato en el archivo de salida
                pw.println( "se guardo el dato " + cont + ":" + dato );

                // Desplegamos el dato en pantalla
                System.out.println( "se guardo el dato " + cont
                    + " : " + dato );
            }

            } catch ( FileNotFoundException e ) {
                e.printStackTrace();
            } catch ( IOException e ) {
                e.printStackTrace();
            } finally {
                if ( pw != null )
                    pw.close();
            }
        }
    }
}
```

Ejercicio 2. Si los datos en el archivo de entrada son, por ejemplo: 10 12 14 16 18, determina, sin ver la solución, cuál será el contenido del archivo de salida generado al correr el programa anterior.

Solución:

```
se guarda el dato 1 : 10
se guarda el dato 2 : 12
se guarda el dato 3 : 14
se guarda el dato 4 : 16
se guarda el dato 5 : 18
```

En caso de que el archivo de entrada pudiera contener no sólo números enteros, debemos incluir bloques `catch` para cada uno de los tipos de excepciones arrojadas por el método `nextInt()`.

Persistencia de objetos mediante serialización

Además de poder leer y escribir en archivos de texto, es importante poder guardar el estado de los objetos para su futuro uso, aún cuando el programa o sistema termine. A este concepto se le conoce como *persistencia*.

La forma de implementar la persistencia varía de acuerdo con el tipo de aplicación que se esté desarrollando, que sería mediante una base de datos, transmisión a un servidor, archivos de entrada/salida, etc. En nuestro caso, utilizaremos archivos binarios.

El uso de archivos binarios es idéntico al uso de archivos de texto, sólo que, como se puede inferir por su nombre, el contenido no será inteligible para nosotros.

Java tiene una forma simple de lograr la persistencia y es a través de la serialización. La *serialización* no es más que la conversión del estado de un objeto a una secuencia de bytes y son precisamente estos bytes los que se escriben o leen desde un archivo binario.

Java provee la interfaz `java.io.Serializable`, la cual debe ser implementada por una clase para que sus instancias puedan ser serializables. Esta interfaz no define métodos ni atributos y sólo sirve para identificar la semántica de ser serializable.

El proceso inverso a la serialización es la *deserialización*, que consiste en leer una secuencia de bytes para recuperar el estado de un objeto.

En general, para especificar que una clase puede ser serializable, se tiene que importar la interfaz `java.io.Serializable` y la clase debe implementar ésta de la siguiente forma:

```
import java.io.Serializable;  
  
public class NombreClase implements Serializable {  
  
    // Definición de la clase  
  
}
```

De la misma forma que se lee y escribe en archivos de texto, Java provee clases para poder leer y escribir en archivos binarios. Las clases `FileInputStream` y `FileOutputStream` son, respectivamente, para leer y escribir en archivos binarios. Una vez que se abre un archivo binario para lectura o escritura, éste debe estar asociado a un objeto que nos permita leer y escribir en el archivo abierto. Esto se logra mediante el uso de las clases `ObjectInputStream` y `ObjectOutputStream`.

Ejemplo 1. Serializar dos objetos, uno tipo `Alumno` y otro tipo `Profesor`, en el archivo binario `datos.bin`.

Antes de poder serializar las instancias de las clases `Alumno` y `Profesor`, necesitamos definirlas como serializables. Debido a que éstas son subclases de la clase `Persona`, basta con definir a superclase como serializable para que sus subclases también lo sean. Esto se logra mediante la implementación de la interfaz `java.io.Serializable` como se muestra enseguida:

```
import java.io.Serializable;  
  
public class Persona implements Serializable {  
  
    // El resto del código como está originalmente  
  
}
```

La clase `Persona` tiene un atributo de tipo `Fecha`, entonces, para que una instancia de la clase `Persona` pueda ser totalmente serializable, *todos sus atributos deben ser objetos de clases serializables*. Definimos a la clase `Fecha` como serializable de la siguiente forma:

```
import java.io.Serializable;  
  
public class Fecha implements Serializable {
```

```
// El resto del codigo como esta originalmente  
}
```

Escribimos ahora una clase principal para crear un objeto **Alumno** y otro **Profesor** y los escribimos en el archivo binario **datos.bin**.

```
import java.io.FileOutputStream;  
import java.io.ObjectOutputStream;  
  
public class EscrituraObjetosMain {  
    public static void main( String[] args ) {  
        ObjectOutputStream oos = null;  
  
        // Se crea un alumno  
        Alumno elAlumno = new Alumno( "Eloy Mata Arce",  
                                      new Fecha( 1990, 2, 1 ),  
                                      "2008112233" );  
        // Se crea un profesor  
        Profesor elProfe = new Profesor("Elmer Homero Petatero",  
                                         new Fecha( 1987, 12, 11 ),  
                                         23245 );  
        try {  
            FileOutputStream fos = new FileOutputStream( "datos.bin" );  
            oos = new ObjectOutputStream( fos );  
  
            // Se escribe el objeto Alumno  
            oos.writeObject( elAlumno );  
  
            // Se escribe el objeto Profesor  
            oos.writeObject( elProfe );  
        }  
        catch ( IOException e ) {  
            throw e;  
        }  
        finally {  
            if ( oos != null ) {  
                oos.close();  
            }  
        }  
    }  
}
```

Ejemplo 2. Deserializar los objetos **Alumno** y **Profesor** del archivo binario **datos.bin**.

En este caso, nosotros sabemos el orden en que fueron guardados los objetos en el archivo `datos.bin`, entonces podemos leer en ese mismo orden los objetos como en el siguiente programa:

```
import java.io.FileInputStream;
import java.io.ObjectInputStream;

public class LecturaObjetosMain {
    public static void main( String[] args ) {
        ObjectInputStream ois = null;

        try {
            FileInputStream fis = new FileInputStream( "datos.bin" );
            ois = new ObjectInputStream( fis );

            // Se lee el objeto Alumno
            Alumno elAlumno = (Alumno) ois.readObject();

            // Se lee el objeto Profesor
            Profesor elProfe = (Profesor) ois.readObject();

            System.out.println( elAlumno );
            System.out.println( elProfe );
        }
        catch ( IOException e ) {
            throw e;
        }
        finally {
            if ( ois != null ) {
                ois.close();
            }
        }
    }
}
```

Ejercicio 3. Determina, sin ver la solución, la salida del programa anterior.

Solución:

```
Eloy Mata Arce nacido el: 1 de Feb de 1990 mat :2008112233
Elmer Homero Petatero nacido el: 11 de Dic de 1987 clave: 23245
```

Ejercicios de entrada/salida de datos

Ejercicio 1. Hacer un programa que use la clase `LectorDeTeclado` para pedir al usuario tres números enteros, a , b y c , y que escriba en pantalla si los números dados son solución o no de la ecuación:

$$a^2+b^2=c^2$$

El método para elevar un número a una potencia en Java es:

```
Math.pow( base, exponente )
```

Solución parcial:

```
public class mainCuadrados {  
  
    public static void main( String[] args ) {  
        Double a;  
        Double b;  
        Double c;  
        LectorDeTeclado in;  
  
        // leer los datos  
        ...  
  
        // Hacer comparacion y enviar mensaje  
        ...  
  
    }  
}
```

Para leer los datos del teclado es necesario instanciar primero el objeto `in`. A continuación presentamos la solución final:

```
public class mainCuadrados {  
  
    public static void main( String[] args ) {  
        Double a;  
        Double b;  
        Double c;  
        LectorDeTeclado in;  
  
        // Leer los datos  
        in = new LectorDeTeclado();  
        System.out.println("a? ");  
        a = in.capturarDoble();  
        System.out.println("b? ");  
        b = in.capturarDoble();  
        System.out.println("c? ");  
        c = in.capturarDoble();  
  
        // Hacer comparacion y enviar mensaje  
        if ( Math.pow( a, 2 ) + Math.pow( b, 2 )  
            == Math.pow( c, 2 ) ) {
```

```
        System.out.println( "Los numeros sí son solucion" );
    }
} else {
    System.out.println( "Los numeros no son solucion" );
}
}
```

Ejercicio 2. Modificar la clase `LectorDeTeclado` de tal forma que reciba como parámetro un mensaje para el usuario. Los métodos que capturan el dato del teclado deben desplegar primero el mensaje en la pantalla para que el usuario sepa qué dato introducir. Modificar la clase para que sea estática y así no sea necesario instanciarla para hacer uso de ella.

Solución:

```
package LectorDeTeclado; // Creamos la clase dentro de un paquete para que ésta pueda usarse desde otros proyectos

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

public class LectorDeTeclado {

    private static InputStreamReader isr =
            new InputStreamReader( System.in );
    private static BufferedReader reader =
            new BufferedReader( isr );

    Quitamos el constructor y declaramos métodos y atributos como estáticos. // Agregamos como parámetro mensaje que es el que se despliega al usuario para pedir el dato.

    public static String leeString( String mensaje ) {
        try {
            System.out.println( mensaje );
            // La entrada finaliza al pulsar Enter
            return reader.readLine();
        } catch ( IOException e ) {
            System.err.println( "Error: 0: " + e.getMessage() );
            return null;
        }
    }

    // Método para capturar enteros
    public static Integer capturarEntero( String mensaje ) {
        try {
            return Integer.parseInt( leeString( mensaje ) );
        }
    }
}
```

```
    } catch ( NumberFormatException e ) {
        System.err.println( "Error 1: " + e.getMessage() );
        return Integer.MIN_VALUE; // valor mas pequeño
    }
}

// Metodo para capturar dobles
public static double capturarDoble( String mensaje ) {
    try {
        return Double.parseDouble( leeString( mensaje ) );
    } catch ( NumberFormatException e ) {
        System.err.println( "Error 3: " + e.getMessage() );
        return Double.NaN // Indica no es un numero;
    }
}
```

Ejercicio 3. Modificar la clase principal del Ejercicio 1 haciendo uso de la nueva clase LectorDeTeclado del Ejercicio 2.

Solución:

```
import LectorDeTeclado.*;

public class mainCuadrados {

    public static void main( String[] args ) {
        Double a;
        Double b;
        Double c;
        LectorDeTeclado in;

        // Leer los datos
        a = LectorDeTeclado.capturarEntero( "a? " );
        b = LectorDeTeclado.capturarEntero( "b? " );
        c = LectorDeTeclado.capturarEntero( "c? " );

        // Hacer comparación y enviar mensaje
        if ( Math.pow( a, 2 ) + Math.pow( b, 2 )
            == Math.pow( c, 2 ) ) {
            System.out.println( "Los numeros si son solucion" );
        }
        else {
            System.out.println( "Los numeros no son solucion" );
        }
    }
}
```

Ejercicio 4. Agregar un método a la clase `LectorDeTeclado` del Ejercicio 2 para capturar un arreglo de tipo `Double`. El tamaño del arreglo es variable y se debe pedir al usuario.

Solución:

```
// Método para capturar arreglos de Double de tamaño variable
public static Double[] capturarArrDoble( String mensaje ) {
    Integer tamanio = capturarEntero(
        "¿cuántos elementos hay en el arreglo?" );

    if ( tamanio < 1 ) {
        return null;
    }
    Double[] datos = new Double[ tamanio ];

    for ( int i = 0; i < datos.length; i++ ) {
        datos[ i ] = capturarDoble( mensaje + "[" + i + "]=" );
    }
    return datos;
}
```

Ejercicio 5. Escribe una clase principal que lea los datos de un arreglo usando el método del *Ejercicio 4* y que despliegue en pantalla los datos del arreglo leído.

Solución:

```
import LectorDeTeclado.*;

public class main {

    public static void main( String[] args ) {

        Double[] x = LectorDeTeclado.capturarArrDoble(
            "Da un número real" );

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

Ejercicio 5.a. Determina, sin ver la solución, la salida del programa anterior cuando los datos capturados con el teclado sean: 4, 44, 12, 9 y 13.

Solución:

```

¿Cuantos elementos hay en el arreglo?
4
Da un número real[0]=
44
Da un número real[1]=
12
Da un número real[2]=
9
Da un número real[3]=
13
x[0]= 44
x[1]= 12
x[2]= 9
x[3]= 13

```

Ejercicio 6. Modificar el Ejemplo 3 de la sección donde se explica cómo escribir datos en un archivo de salida, para que escriba los datos leídos del archivo de entrada en un orden invertido.

Solución:

```

import java.util.Scanner;
import java.io.*;

public class ArchivosMain {

    private static int[] A;

    public static void main( String[] args ) {

        File archivoEntrada = new File( "datos.txt" );
        FileWriter archivoSalida = null;
        PrintWriter pw = null;

        int cont = 0;
        int dato;
        A = new int[15];
        try {
            Scanner scanner = new Scanner( archivoEntrada );
            archivoSalida = new FileWriter( "salida.txt" );
            pw = new PrintWriter( archivoSalida );

            while ( scanner.hasNextLine() ) {
                dato = scanner.nextInt();
                A[cont] = dato;
                cont++;
            }
        } catch ( Exception e ) {
            e.printStackTrace();
        }
    }
}

```

Guardamos los datos leídos en un arreglo. Los vamos contando en cont.

```
for ( int i = cont - 1; i >= 0; i-- )  
    pw.println( A[i] ); ←  
  
} catch( FileNotFoundException e ) {  
    e.printStackTrace();  
  
} catch( IOException e ) {  
    e.printStackTrace();  
  
} finally {  
    if ( archivoSalida != null )  
        archivoSalida.close();  
}  
}  
}
```

Para guardar los datos en el archivo recorremos el arreglo en orden invertido.

Ejercicio 6.a. Determina, sin ver la solución, el contenido del archivo de salida al correr el programa anterior si los datos en el archivo de entrada son: 10 12 14 16 18.

Solución:

18
16
14
12
10

Arreglos en Java (*listas homogéneas*)

En Java, los arreglos son objetos y son instancias de una clase especial denominada por `[]`. La sintaxis para la declaración de un objeto de clase arreglo es la siguiente:

`tipoElemento[1] NombreArreglo:`

La sintaxis para crear un arreglo es:

```
NombreArreglo = new tipoElemento[dimension];
```

De lo anterior, se observa que la dimensión se especifica hasta que se crea el objeto. A continuación damos unos ejemplos de declaración de arreglos en Java:

```
Float[] arregloDeFloats;  
Integer[] arregloDeEnteros;
```

```
Boolean[] arregloDeBooleans;
String[] arregloDeStrings;
Alumno[] arregloDeAlumnos;
```

Para crear estos arreglos tenemos, por ejemplo:

```
arregloDeFloats = new Float[200];
arregloDeEnteros = new Integer[1500];
arregloDeBooleans = new Boolean[35];
arregloDeStrings = new String[1000];
arregloDeAlumnos = new Alumno[350];
```

Como los arreglos en Java son objetos, éstos tienen atributos que proporcionan información del objeto. El atributo `length` regresa como resultado el tamaño del arreglo.

Ejemplo 1. Declarar un arreglo de enteros de tamaño 30 y poner en cada uno de sus elementos su propio índice multiplicado por dos. Imprimir cada uno de los elementos del arreglo.

Solución:

```
public class Main {

    public static void main( String[] args ) {

        Integer[] arregloDeEnteros; // declaracion del arreglo

        arregloDeEnteros = new Integer[30]; // instancia del arreglo

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

Los elementos de los arreglos en Java son objetos.

Ejemplo 2. Haremos un arreglo con 3 objetos de la siguiente clase `Alumno`:

```
public class Alumno {

    String nombre;
    Double calific;
```

```
Integer matricula;

public Alumno( String nom, Double cal, Integer matr ) {
    nombre = nom;
    calific = cal;
    matricula= matr;
}

public String toString() {
    return nombre
        + " matricula: " + matricula
        + " tiene: " + calific;
}
}
```

Solución parcial:

```
public class Main {  
  
    public static void main( String[] args ) {  
  
        // Declaración del arreglo  
        ...  
  
        // instancia del arreglo  
        ...  
  
        // Los 3 objetos del arreglo son los siguientes  
        alumnos[0] = new Alumno( "Pedro Martinez", 9.8, 943012 );  
        alumnos[1] = new Alumno( "Eleazar Hernandez", 8.5, 274901 );  
        alumnos[2] = new Alumno( "Obdulia Garcia", 7.3, 204117 );  
  
        // Desplegar en pantalla cada elemento del arreglo  
        ...  
    }  
}
```

Solución:

```
public class Main {  
  
    public static void main( String[] args ) {  
  
        // Declaración del arreglo  
        Alumno[] alumnos;  
  
        // instancia del arreglo  
        alumnos = new Alumno[3]; // se instancian las referencias  
                                // a los objetos
```

```
// Los 3 objetos del arreglo son los siguientes
alumnos[0] = new Alumno( "Pedro Martinez", 9.8, 943012 );
alumnos[1] = new Alumno( "Eleazar Hernandez", 8.5, 274901 );
alumnos[2] = new Alumno( "Obdulia Garcia", 7.3, 204117 );

// Desplegar en pantalla cada elemento del arreglo
for ( int i = 0; i < alumnos.length; i++ ) {
    System.out.println( alumnos[i] );
}
```

Arreglos como parámetros de entrada a un método

Los arreglos pueden pasarse como parámetros de entrada a un método. Por ejemplo, si tenemos el arreglo:

```
Double[] temperaturas = new Double[5];
```

El llamado a un método que recibe un arreglo como parámetro se hace de la siguiente manera:

```
despliegaArreglo( temperaturas );
```

Es decir, basta con enviar como parámetro actual el nombre del arreglo, el cual *es una referencia* al inicio del arreglo. El método que recibe el arreglo debe tener declarado el arreglo como parámetro formal con la siguiente sintaxis:

```
public tipo nombreMetodo( tipoElemento[] nombreArreglo)
```

En el caso de nuestro ejemplo, tendríamos:

```
public void despliegaArreglo( Double[] A ) {
    for ( int i = 0; i < A.length; i++ ) {
        System.out.println( "temperatura " + i + " =" + A[i] + " " );
    }
}
```

El hecho de que el método reciba una referencia al arreglo, implica que los cambios que el método lleve a cabo los hace directamente sobre el arreglo original. Por lo tanto, cuando se envía un arreglo a un método no es necesario regresarlo como valor de retorno. Sin embargo, puede haber casos en los que sea necesario regresar un arreglo como valor de retorno desde un método. Esto se expone a continuación.

Arreglos como valor de retorno de un método

También es posible que un método regrese un arreglo, en este caso la sintaxis para el método es la siguiente:

```
public tipoElemento[] nombreMetodo( parametros )
```

Cuando se invoca a un método que regresa un arreglo, se debe hacer la asignación en una variable de tipo arreglo, por ejemplo, en la variable `datosTemperatura`. Supongamos que se preguntó previamente por el valor de `tamanio`.

```
Double[] datosTemperatura = new Double[ tamanio ];
```

Entonces, la llamada a un método que captura los datos del arreglo y regresa el arreglo capturado se hace de la siguiente forma:

```
datosTemperatura = capturarArreglo( tamanio );
```

A continuación presentamos el programa completo, que también despliega los valores del arreglo. La clase `LectorDeTeclado` es la que presentamos en los “Ejercicios de pilas” del cuarto capítulo.

```
public class ArregloMain {  
  
    public static void main( String[] args ) {  
        Integer tamanio;  
  
        tamanio = LectorDeTeclado.capturarEntero(  
                "De que tamanio es el arreglo?");  
        Double[] datosTemperatura = new Double[ tamanio ];  
  
        datosTemperatura = capturarArreglo( tamanio );  
        desplegarArreglo( datosTemperatura );  
    }  
  
    // Método para capturar los valores de un arreglo  
    public static Double[] capturarArreglo( int tamanio ) {  
  
        Double[] datos = new Double[ tamanio ];  
  
        for ( int i = 0; i < tamanio; i++ ) {  
            datos[i] = LectorDeTeclado.capturarEntero(  
                    "Dato " + i + " ? ");  
        }  
  
        return datos; // regresa la referencia al arreglo
    }
}
```

```
}

// Método para desplegar los valores de un arreglo
public static void despliegaArreglo( Double[] A ) {

    for ( int i = 0; i < A.length; i++ )
        System.out.println( "temperatura " + i + " = " + A[i] + " " );
}

}
```

Ejercicios con arreglos

Ejercicio 1. Hacer las siguientes declaraciones.

- 1.1 Declarar e instanciar un arreglo de 17 `Integer` llamado A.
- 1.2 Declarar e instanciar un arreglo de 1150 `Double` llamado B.
- 1.3 Declarar e instanciar un arreglo de 300 `Float` llamado C.
- 1.4 Declarar e instanciar un arreglo de 3 elementos de la clase `Manecilla` llamado `manecillas`.

Soluciones:

- 1.1 `Integer[] A = new Integer[17];`
- 1.2 `Double[] B = new Double[1150];`
- 1.3 `Float[] C = new Float[300];`
- 1.4 `Manecilla[] manecillas = new Manecilla[3];`

Ejercicio 2. Suponiendo que ya tenemos un arreglo de `int` llamado A, de tamaño n, ¿cómo haces el código que pide al usuario cada uno de sus elementos?

Solución:

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

for ( int i = 0; i < n; i++ ) {
    System.out.println( "A[" + i + "]=?" );
    A[i] = input.nextInt();
}
```

Nótese que no tuvimos que codificar nuestra propia clase.

Ejercicio 3. Suponiendo que ya tenemos un arreglo de `Integer` llamado B, ¿cómo haces el código que pide al usuario cada uno de sus elementos?

Solución:

```
LectorDeTeclado input = new LectorDeTeclado();  
  
for ( int i = 0; i < B.length; i++ )  
    B[i] = input.leeInteger( "B[" + i + "]=?");
```

La ventaja de hacer nuestra propia clase es que se ajusta a nuestras necesidades, como en este caso la clase `LectorDeTeclado`.

Ejercicio 4. Si ya tenemos los datos del arreglo B, ¿cómo haces el código que despliega en la pantalla cada uno de sus elementos?

Solución:

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

Ejercicio 5. Ahora agrega a la clase `LectorDeTeclado` el nuevo método llamado `leerIntegerArray`, que regrese un arreglo de `Integer` capturados del teclado y que reciba como parámetro el tamaño del arreglo a capturar y el texto que saldrá en pantalla para cada captura.

Solución:

```
public Integer[] leerIntegerArray(  
        Integer tamanio, String pregunta ) {  
  
    Integer[] datos = new Integer[ tamanio ];  
  
    for ( int i = 0; i < tamanio; i++ ) {  
        datos[i] = capturarEntero( pregunta + "[" + i + "]" );  
    }  
  
    return datos;  
}
```

Ejercicio 6. ¿Cómo haces el llamado al método `leerIntegerArray` para que capture los datos de un arreglo de 10 elementos llamado A?

Solución:

```
Integer[] A;
A = leeIntegerArray( 10, "A" );
```

Ejercicio 7. Si tenemos la clase **Alumno** definida como:

```
public class Alumno extends Persona {

    private String matricula;

    public Alumno( String n, Fecha f, String m ) {
        super( n, f );
        matricula = m;
    }

    public String toString() {
        return super.toString() + " mat:" + matricula;
    }
}
```

Ejercicio 7.1. ¿Cómo construyes un arreglo llamado **alumnos** de 5 elementos de clase **alumno**?

Solución:

```
Alumno[] alumnos = new Alumno[5];
```

Ejercicio 7.2. ¿Cómo haces un método que reciba un **Integer** con el tamaño del arreglo, que genere un arreglo de **Alumno** del tamaño indicado, que capture del teclado cada elemento del arreglo **alumnos** y que devuelva el arreglo capturado? Llamaremos al método **capturaAlumnos**.

Solución:

```
public static Alumno[] capturaAlumnos( Integer size ) {

    Alumno[] datos = new Alumno[ size ];

    for ( int i = 0; i < size; i++ ) {
        datos[i] = ← Debemos instanciar cada objeto del
                  new Alumno(   arreglo
                               input.capturarLinea(
                                   "¿Cuál es el nombre del alumno" + (i + 1) + "?"),
                               new Fecha(
```

```

        LectorDeTeclado.capturarEntero(
            "¿Cuál es su año de nacimiento?"),
        LectorDeTeclado.capturarEntero(
            "¿Cuál es su mes de nacimiento?"),
        LectorDeTeclado.capturarEntero(
            "¿Cuál es su día de nacimiento?")
        ),
        LectorDeTeclado.capturarEntero("¿Su matrícula?")
    );
}

return datos;
}

```

El dato que el usuario da desde el teclado se envía directamente al constructor de la clase Alumno

Se instancia un objeto de clase Fecha y se envía como segundo parámetro del constructor de la clase Alumno.

Ejercicio 7.3 ¿Como llamas al método **capturaAlumnos** para guardar los datos capturados en un arreglo de 100 elementos que se llame **alumnos**?

Solución:

```

Alumno[] alumnos;
alumnos = capturaAlumnos( 100 );

```

Ejercicio 7.4. ¿Cómo haces un método que despliegue cada elemento del arreglo **alumnos**? El método deberá recibir el arreglo como parámetro. Llamaremos al método **desplegarAlumnos**.

Solución:

```

public static void desplegarAlumnos( Alumno[] a ) {

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

```

Como se observa, el arreglo de objetos se maneja de la misma manera que cualquier otro arreglo. Cuando se requiere imprimir el objeto, se invoca automáticamente a su método **toString()**, el cual debe estar definido previamente.

Ejercicio 7.5. ¿Cómo llamas a **desplegarAlumnos()** desde el método **main**?

Solución:

```
desplegarAlumnos( alumnos );
```

Ejercicio 8. En este ejercicio utilizaremos las siguientes clases:

La clase **Temperatura** guarda la temperatura en su único atributo y tiene los métodos necesarios para hacer la conversión de Farenheit a Centígrados y viceversa:

```
public class Temperatura {

    private Double centigrados;

    Temperatura() {
        centigrados = -273.0; // inicializa con el valor del cero
                             // absoluto
    }

    public void setCentigrados( Double c ) {
        centigrados = c;
    }

    public void setEnFarenheit( Double f ) {
        centigrados = 5 / 9.0 * (f - 32); ←
    }

    public Double getCentigrados() {
        return centigrados;
    }

    public Double getFarenheit() { ←
        return 9 / 5.0 *centigrados + 32;
    }
}
```

Si se recibe el dato en Farenheit, se hace la conversión F → C.

Si se requiere el dato en Farenheit, se hace la conversión C → F.

La clase *abstracta* **Menu** está preparada para desplegar un menú de opciones, validar que el usuario proporcione un número que se encuentre dentro de la lista de opciones y regresar la opción seleccionada por el usuario:

```
package Utilerias; // la pondremos dentro de un paquete hecho
                  // por nosotros

public abstract class Menu {

    // Las clases descendientes de Menu deberan inicializar
    // las opciones.
    // opcion[0] es la opción 1 del menu en pantalla.

    protected String[] opciones;

    public Integer opcion() {
        if ( opciones.length < 1 ) {
            return 0;
        }
    }
}
```

```

System.out.println();
for ( int i = 0; i < opciones.length; i++ ) {
    System.out.println( (I + 1) + ":" + opciones[i] );
}
System.out.println();

Integer opcion;
do {
    opcion = LectorDeTeclado.capturarEntero(
        "Escriba la opción deseada:" );
} while ( opcion < 1 || opcion > opciones.length );

return opcion;
}
}

```

La clase `Menu`, define al atributo `opciones` como `protected`. Esto quiere decir que este atributo es público para todas las subclases de `Menu`, pero privado para el resto de las clases.

La clase `MenuTemperaturas`, hija de `Menu`, define que serán 5 opciones y cual será cada una de estas opciones:

```

import Utilerias.Menu;

public class MenuTemperaturas extends Menu {

    MenuTemperaturas() {
        opciones = new String[5];
        opciones[0] = "Capturar en Centigrados";
        opciones[1] = "Capturar en Farenheit";
        opciones[2] = "Desplegar en Centigrados";
        opciones[3] = "Desplegar en Farenheit";
        opciones[4] = "Salir";
    }
}

```

Ejercicio 8.1. Crear la clase `TemperaturaArray`, que tiene como atributo:

- `t` de tipo arreglo de `Temperatura`

Y tiene como métodos:

- `TemperaturaArray(Integer n)`. El constructor instancia el arreglo de temperaturas.

- `capturarEnCentigrados()`. Captura las temperaturas en grados centígrados y las guarda en el arreglo usando el método apropiado de la clase `Temperatura`.
- `capturarEnFarenheit()`. Captura las temperaturas en grados Fahrenheit y las guarda en el arreglo usando el método apropiado de la clase `Temperatura`.
- `desplegarEnCentigrados()`. Despliega en pantalla las temperaturas en grados centígrados.
- `desplegarEnFarenheit()`. Despliega en pantalla las temperaturas en grados centígrados.

Primera Solución parcial

```
import Utlcerias.LectorDeTeclado;

public class TemperaturaArray {

    Temperatura[ ] t;

    TemperaturaArray( Integer n ) {
        t = new Temperatura[n];

        for ( int i = 0; i < n; i++ ) {
            t[i] = new Temperatura();
        }
    }

    public void CapturarEnCentigrados() {
        ...
    }

    public void CapturarEnFarenheit() {
        ...
    }

    public void DesplegarEnCentigrados() {
        ...
    }

    public void DesplegarEnFarenheit() {
        ...
    }
}
```

Segunda solución parcial

```
import Utiles.LectorDeTeclado;

public class TemperaturaArray {

    Temperatura[] t;

    TemperaturaArray( Integer n ) {
        t = new Temperatura[n];

        for ( int i = 0; i < n; i++ ) {
            t[i] = new Temperatura();
        }
    }

    public void CapturarEnCentigrados() {
        for ( int i = 0; i < t.length; i++ )
            t[i].setCentigrados(
                LectorDeTeclado.capturarDoble(
                    "t[" + i + "] en Centígrados: " ));
    }

    public void CapturarEnFarenheit() {
        ...
    }

    public void DesplegarEnCentigrados() {
        for ( int i = 0; i < t.length; i++ )
            System.out.println( t[i].getCentigrados() + "°C" );
    }

    public void DesplegarEnFarenheit() {
        ...
    }
}
```

Solución final:

```
import Utiles.LectorDeTeclado;

public class TemperaturaArray {

    Temperatura[] t;

    TemperaturaArray( Integer n ) {
        t = new Temperatura[n];

        for ( int i = 0; i < n; i++ ) {
```

```

        t[i] = new Temperatura();
    }

}

public void CapturarEnCentigrados() {
    for ( int i = 0; i < t.length; i++ )
        t[i].setCentigrados(
            LectorDeTeclado.capturarDoble(
                "t[" + i + "] en Centígrados: " ));
}

public void CapturarEnFarenheit() {
    for ( int i = 0; i < t.length; i++ )
        t[i].setEnFarenheit(
            LectorDeTeclado.capturarDoble(
                "t[" + i + "] en Farenheit: " ));
}

public void DesplegarEnCentigrados() {
    for ( int i = 0; i < t.length; i++ )
        System.out.println( t[i].getCentigrados() + "°C" );
}

public void DesplegarEnFarenheit() {
    for ( int i = 0; i < t.length; i++ )
        System.out.println( t[i].getFarenheit() + "°F" );
}
}

```

Ejercicio 8.2. Completar la siguiente clase principal de tal manera que el usuario tenga la opción de introducir una lista de 5 temperaturas dadas en grados Centígrados o Farenheit, la opción de desplegar en pantalla estas temperaturas en grados Centígrados o Farenheit y una opción para salir del programa:

```

import Utilerias.LectorDeTeclado;

public class TemperaturasMain {

    public static void main( String[] args ) {

        TemperaturaArray t = ... // instanciar el objeto con tamaño 5
        MenuTemperaturas menu = ... // instanciar el objeto menu

        while( true ) {
            Integer opcion = ...// hacer que se despliegue el menu
                // y se capture una opcion valida

            switch( opcion ) {
                case 1: ... // captura en centigrados

```

```
        case 2: ... // captura en Farenheit  
        case 3: ... // desplegar en centigrados  
        case 4: ... // desplegar en Farenheit  
        case 5: System.out.println( "adios" ); return;  
    }  
}  
}  
}
```

Solución parcial:

```
import Utileria.lectorDeTeclado;

public class TemperaturasMain {

    public static void main( String[] args ) {

        TemperaturaArray t = new TemperaturaArray( 5 );
        MenuTemperaturas menu = new MenuTemperaturas();

        while( true ) {
            Integer opcion = ... // hacer que se despliegue el menu
                                // y se capture una opcion valida

            switch( opcion ) {
                case 1: ... // captura en centigrados
                case 2: ... // captura en Farenheit
                case 3: ... // desplegar en centigrados
                case 4: ... // desplegar en Farenheit
                case 5: System.out.println( "adios" );      return;
            }
        }
    }
}
```

Solución final:

```
import Utilerias.LectorDeTeclado;

public class TemperaturasMain {

    public static void main( String[] args ) {

        TemperaturaArray t = new TemperaturaArray( 5 );
        MenuTemperaturas menu = new MenuTemperaturas();

        while( true ) {
            Integer opcion = menu.opcion();
```

```

        switch( opcion ) {
            case 1: t.capturarEnCentigrados();      break;
            case 2: t.capturarEnFarenheit();       break;
            case 3: t.desplegarEnCentigrados();    break;
            case 4: t.desplegarEnFarenheit();     break;
            case 5: System.out.println( "adios" ); return;
        }
    }
}
}

```

Listas con objetos

Concepto de lista ligada

Las listas ligadas se forman con “cajas” que contienen un elemento y un enlace a la siguiente caja, como se muestra en la figura IV-1.

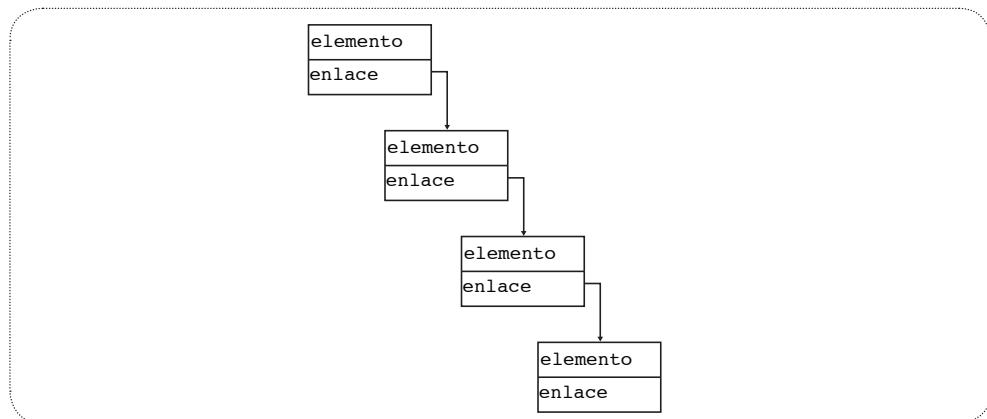


Figura IV-1. Lista ligada

Ya existe en Java una clase que se llama **LinkedList**. La reutilización de código permite trabajar con listas ligadas sin tener que programar todo su funcionamiento. Para poder hacer uso de la clase **LinkedList** es necesario importarla.

```
import java.util.LinkedList;
```

El elemento que contiene la lista ligada es un objeto de cualquier clase, incluyendo las wrapper (**Integer**, **String**, **Double**, ...). La sintaxis para declarar e instanciar una lista ligada de una **ClaseX**, es la siguiente:

```
LinkedList<ClaseX> nombreLista;  
nombreLista = new LinkedList<ClaseX>();
```

O todo en un solo renglón:

```
LinkedList<ClaseX> nombreLista = new LinkedList<ClaseX>();
```

Al indicar entre los signos <> la clase a utilizar, estamos estableciendo que esta lista ligada contendrá exclusivamente elementos de ese tipo. De esta forma, si queremos insertar elementos que no correspondan al tipo especificado, el compilador de Java lo detectará y nos enviará un mensaje de error. Lo mismo sucede si lo que queremos es recuperar un elemento de la lista y hacer la referencia a él con un tipo que no corresponde.

Esta clase tiene algunos métodos de mucha utilidad, por ejemplo, podemos añadir un elemento al principio de la lista ligada con el método `addFirst()` o podemos añadir un elemento al final con el método `addLast()`.

Para obtener un elemento *i* de la lista ligada usamos el método `get(i)`, como se muestra en la figura IV-2.

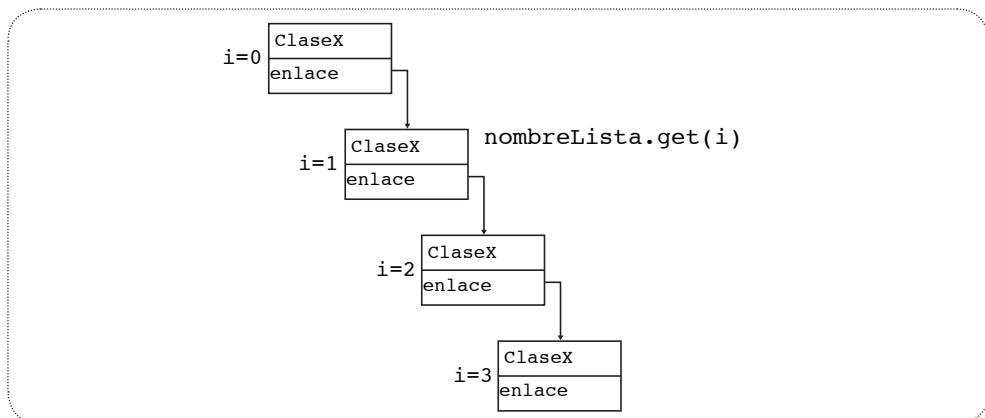


Figura IV-2. Obtención de un elemento de la lista ligada

A continuación presentamos el código para construir una lista ligada de `Integer`.

```
import java.util.LinkedList;

public class ListasLigadasMain {

    public static void main( String[] args ) {

        // Se declara e instancia una lista de Integer
        LinkedList<Integer> listaEnteros = new LinkedList<Integer>();

        //Añadimos 3 elementos al principio
    }
}
```

```
listaEnteros.addFirst( 1 );
listaEnteros.addFirst( 2 );
listaEnteros.addFirst( 3 );

//Añadimos un elemento al final
listaEnteros.addLast( 0 );

desplegarLista( listaEnteros );
}

public static void desplegarLista( LinkedList l ) {
    for ( int i = 0; i < l.size(); i++ )
        System.out.println( l.get( i ) );
}

}
```

Ejercicio 9. Determina, sin ver la solución, cuál será la salida del programa anterior.

Solución:

```
3
2
1
0
```

Ejercicios con listas ligadas

Ejercicio 1. Hacer una lista ligada de cadenas de caracteres con los siguientes elementos:

- Poner al principio: “Pedro”
- Poner al principio: “María”
- Poner al principio: “Juan”
- Poner al final: “Alicia”

Solución:

```
import java.util.LinkedList;

public class PilasyColasMain {

    public static void main( String[] args ) {

        // hacer una lista de String
        LinkedList<String> listaString = new LinkedList<String>();
```

```
listaString.addFirst( "Pedro" );
listaString.addFirst( "María" );
listaString.addFirst( "Juan" );
listaString.addLast( "Alicia" );

desplegarLista( listaString );
}

public static void desplegarLista( LinkedList l ) {
    for ( int i = 0; i < l.size(); i++ )
        System.out.println( l.get( i ) );
}
}
```

Ejercicio 1.a. Determina, sin ver la solución, cuál será la salida del programa anterior.

Solución:

```
Juan
María
Pedro
Alicia
```

Ejercicio 2. Hacer una lista ligada con objetos de la clase Alumno. Insertar al principio de la lista los siguientes elementos:

- Pedro Martínez matricula: 943012 tiene: 9.8
- Eleazar Hernández matricula: 274901 tiene: 8.5
- Obdulia García matricula: 204117 tiene: 7.3

A continuación definimos la clase Alumno:

```
public class Alumno {

    String    nombre;
    Double   calific;
    Integer  matricula;

    public Alumno( String nom, Double cal, Integer matr ) {
        nombre = nom;
        calific = cal;
        matricula= matr;
    }

    public String toString() {
        return nombre
            + " matricula: " + matricula
    }
}
```

```

        + " tiene: " + calific;
    }
}

```

Solución:

```

import java.util.LinkedList;

public class ListasLigadasMain {

    public static void main( String[] args ) {

        // hacer una lista de Alumnos
        LinkedList<Alumno> listaAlumnos = new LinkedList<Alumno>();

        Alumno alumno1 = new Alumno( "Pedro Martinez", 9.8, 943012 );
        listaAlumnos.addFirst( alumno1 );

        Alumno alumno2 = new Alumno( "Eleazar Hernandez", 8.5, 274901 );
        listaAlumnos.addFirst( alumno2 );

        Alumno alumno3 = new Alumno( "Obdulia Garcia", 7.3, 204117 );
        listaAlumnos.addFirst( alumno3 );

        // desplegar la lista
        desplegarLista( listaAlumnos );
    }
}

```

Primero se instancia cada uno de los objetos y luego se agregan a la lista.

Ejercicio 3. Una lista Last In First Out (LIFO) se caracteriza porque el primer elemento en entrar es el último en salir. Cada vez que se inserta un nuevo elemento, se hace al principio de la lista, de tal forma que el primer elemento disponible es el último que se introdujo.

Hacer un programa que pida datos de tipo entero y que los añada a una lista ligada LIFO, es decir, debe añadir los datos siempre al principio. Se debe preguntar a usuario si desea introducir un dato (0 = no y 1 = si). Cuando el usuario contesta que si, el programa solicita un entero y lo agrega a la lista. Cuando el usuario ya no desea introducir más datos en la lista, se despliegan todos los datos existentes.

Solución:

```

import java.util.LinkedList;
import Utilerias.LectorDeTeclado; //Paquete con la clase

public class LifoMain {

```

```

public static void main( String[] args ) {

    Integer otroElemento;
    Integer entero;
    LinkedList<Integer> listaEnteros = new LinkedList<Integer>();

    do {
        do {
            otroElemento = LectorDeTeclado.capturarEntero(
                "tienes un elemento para la lista?\n"
                + "0 = no      1 = si");
        } while (( otroElemento != 0 )
            && ( otroElemento != 1 ));    //validar entrada

        // Se introduce un elemento sólo si el usuario dijo que si
        if ( otroElemento == 1 ) {
            entero = LectorDeTeclado.capturarEntero( "Dame un entero" );
            listaEnteros.addFirst( entero );
        }
        else {
            desplegarLista( listaEnteros );
        }
    } while ( otroElemento == 1 );
}

public static void desplegarLista( LinkedList l ) {
    for ( int i = 0; i < l.size(); i++ )
        System.out.println( l.get( i ) );
}
}

```

Ejercicio 4. Construir un programa en el que se pueda agregar un objeto a la lista mediante un menú. La lista ligada será de objetos de la clase Alumno del *Ejercicio 2*. La lista ligada de objetos de la clase Alumno. Se llama listaAlumno. El menú se llama **menuListas**, con las siguientes opciones:

- opciones[0] = "Agregar elemento al principio";
- opciones[1] = "Agregar elemento al final";
- opciones[2] = "Desplegar todos los elementos de la lista";
- opciones[3] = "Salir";

Hacer la clase **MenuListas** hija de la clase abstracta **Menu**.

```

Import Utilerias.LectorDeTeclado;

public abstract class Menu {
    // las clases descendientes de Menu deberan
}

```

```
// inicializar las opciones
// opcion[0] es la opción 1 del menu en pantalla.

protected String[] opciones;

public Integer opcion() {
    if ( opciones.length < 1 )
        return 0;

    System.out.println();
    for ( int i = 0; i < opciones.length; i++ )
        System.out.println( i+1 + ":" + opciones[i] );

    System.out.println();

    Integer opcion;
    do {
        opcion = LectorDeTeclado.capturarEntero(
            "Escriba la opción deseada:" );
    } while( opcion < 1 || opcion > opciones.length );

    return opcion;
}
}
```

Solución:

Pondremos las clases `LectorDeTeclado` y `Menu` en un paquete llamado `Utilerias`. La clase `MenuListas` hija de la clase abstracta `Menu` es la siguiente:

```
import Utilerias.Menu;

public class MenuListas extends Menu {

    MenuListas() {
        opciones = new String[4];
        opciones[0] = "Aregar elemento al principio";
        opciones[1] = "Aregar elemento al final";
        opciones[2] = "Desplegar todos los elementos de la lista";
        opciones[3] = "Salir";
    }
}
```

Primera solución parcial:

```
import Utilerias.*;

public class ListaLigadaMain {

    public static void main( String[] args ) {

        LinkedList<Alumno> listaAlumnos = ... // instanciar la lista
        Alumno objeto;
        MenuListas menu = ... // instanciar el menu

        while ( true ) {
            Integer opcion = ... // pedir la opcion al menu
            switch( opcion ) {
                case 1: objeto = capturarAlumno();
                    ... //agregar elemento al inicio de la lista;
                    break;
                case 2: objeto = capturarAlumno();
                    ... //agregar elemento al final de la lista;
                    break;
                case 3: desplegarLista( ... );
                    break;
                case 4: System.out.println( "adios" );
                    return;
            }
        }
    }

    public static void desplegarLista(...) {
        ...
    }

    public static Alumno capturarAlumno() {
        return new Alumno(...,
            ...,
            ...);
    }
}
```

Segunda solución parcial:

```
import Utilerias.*;

public class ListaLigadaMain {

    public static void main( String[] args ) {

        LinkedList<Alumno> listaAlumnos = new LinkedList<Alumno>();
```

```
Alumno objeto;
MenuListas menu = new MenuListas();

while( true ) {
    Integer opcion = menu.opcion();
    switch( opcion ) {
        case 1: objeto = capturarAlumno();
                  listaAlumnos.addFirst( objeto );
                  break;
        case 2: objeto = capturarAlumno();
                  listaAlumnos.addLast( objeto );
                  break;
        case 3: desplegarLista( listaAlumnos );
                  break;
        case 4: System.out.println( "adios" );
                  return;
    }
}

public static void desplegarLista(...) {
    ...
}

public static Alumno capturarAlumno() {
    return new Alumno(...,
                      ...,
                      ...);
}
}
```

Solución final:

```
import Utererias.*;
public class ListaLigadaMain {

    public static void main( String[] args ) {

        LinkedList<Alumno> listaAlumnos = new LinkedList<Alumno>();
        Alumno objeto;
        MenuListas menu = new MenuListas();

        while( true ) {
            Integer opcion = menu.opcion();
            switch( opcion ) {
                case 1: objeto = capturarAlumno();
                          listaAlumnos.addFirst( objeto );
                          break;
```

```

        case 2: objeto = capturarAlumno();
                  listaAlumnos.addLast( objeto );
                  break;
        case 3: desplegarLista( listaAlumnos );
                  break;
        case 4: System.out.println( "adios" );
                  return;
            }
        }
    }

public static void desplegarLista( LinkedList l ) {
    for ( int i = 0; i < l.size(); i++ )
        System.out.println( l.get( i ) );
}

public static Alumno capturarAlumno() {
    return new Alumno(
        LectorDeTeclado.capturarLinea( "Nombre?" ),
        LectorDeTeclado.capturarDoble( "Calificacion?" ),
        LectorDeTeclado.capturarEntero( "Matricula?" )
    );
}
}

```

Clases para pilas y colas

Concepto de pila. En una *pila* se almacenan elementos uno sobre otro, de tal forma que el primer elemento disponible es el último que fue almacenado. A una pila también se le conoce como lista LIFO (Last In First Out), es decir, el primero en entrar es el último en salir. Una pila en la programación puede compararse con una pila de platos o de tortillas. Cada plato (o tortilla) nuevo se coloca hasta arriba de la pila. De esta forma, si deseamos quitar un plato, el primero disponible resulta ser el último que se guardó en la pila de platos.

Existen dos operaciones básicas para trabajar con pilas, que son `push()` y `pop()`. La operación `push()`, que en inglés significa *empujar*, se encarga de poner hasta arriba de la pila un elemento nuevo. En la figura IV-3 se ilustra una pila en la que se hicieron cuatro operaciones `push()` en el siguiente orden:

- `push(elemento1)`
- `push(elemento2)`
- `push(elemento3)`
- `push(elemento4)`

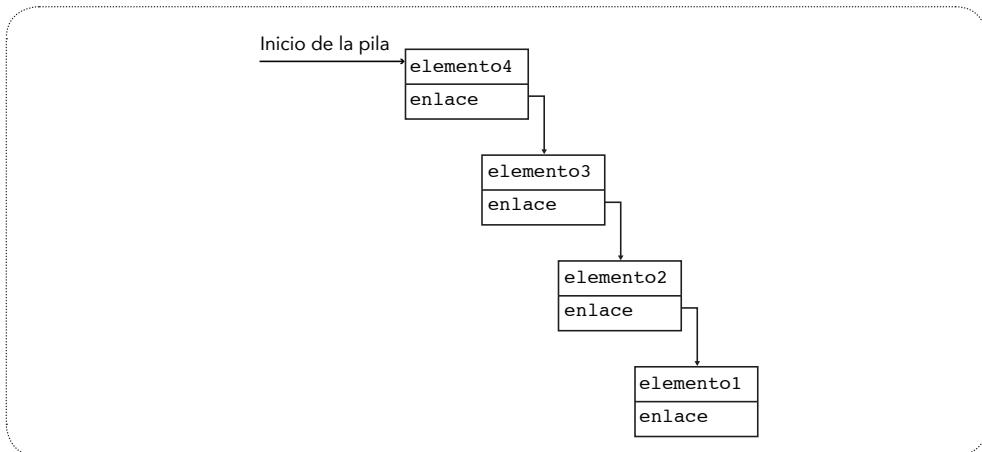


Figura IV-3. Pila: se hizo la operación `push()` a cuatro elementos

La operación `pop()` en este contexto puede traducirse como *expulsar* o *sacar* de la pila. `pop()` entrega un elemento *expulsado* de la pila y el inicio de la pila apunta ahora al siguiente elemento. Como se ilustra en la figura IV-4, el último elemento que entró (elemento4) fue el primero en salir

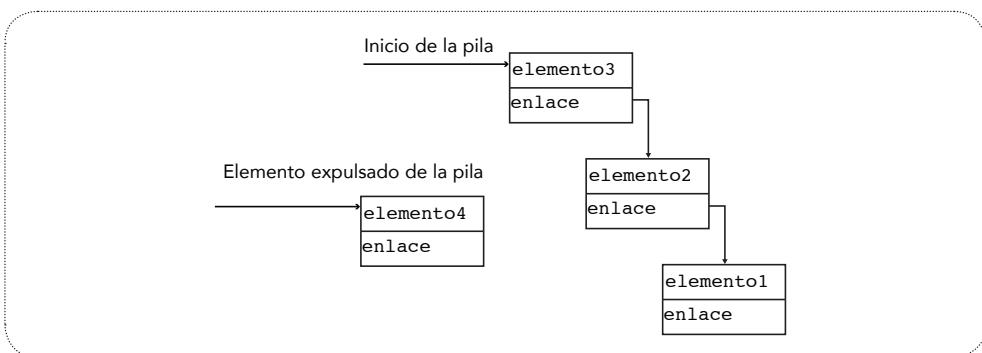


Figura IV-4. Pila: se hizo una operación `pop()` a una pila con cuatro elementos

Ejercicios de pilas

Ejercicio 1. Hacer un programa que inserte 4 enteros a una lista ligada. Posteriormente, debe sacar cada uno de los elementos de la lista y guardarlos en una pila. Finalmente, debe sacar los elementos de la pila e imprimirlos.

Solución:

```

import java.util.*;
public class PilaMain {
    public static void main( String[] args ) {
        Stack<Integer> pila = new Stack<Integer>();
        LinkedList<Integer> lista = new LinkedList<Integer>();

        lista.addFirst( 1 );
        lista.addFirst( 2 );
        lista.addFirst( 3 );
        lista.addFirst( 4 );
    }

    Integer elemento;
    for( int i = 0; i < 4; i++ ) {
        elemento = lista.get( i );
        pila.push( elemento );
    }

    System.out.println( "elementos de la pila: " );
    for( int j = 1; j <= 4; j++ ) {
        elemento = ( Integer ) pila.pop();
        System.out.println( "elemento " + j + " = " + elemento );
    }
}
}

```

Saca un elemento de la lista y lo guarda en la pila.

Saca un elemento de la pila y lo muestra en pantalla.

Ejercicio 1.a. Determina, sin ver la solución, cuál será la salida del programa anterior.

Solución:

```

elementos de la pila:
elemento 1 = 1
elemento 2 = 2
elemento 3 = 3
elemento 4 = 4

```

Para entender mejor este resultado, mostramos en la figura IV-5 la estructura de la lista ligada y de la pila. Como los números se añaden al principio de la lista, estos quedan en orden inverso, al ir sacando los elementos de la lista comenzando por el principio, el orden se vuelve a invertir.

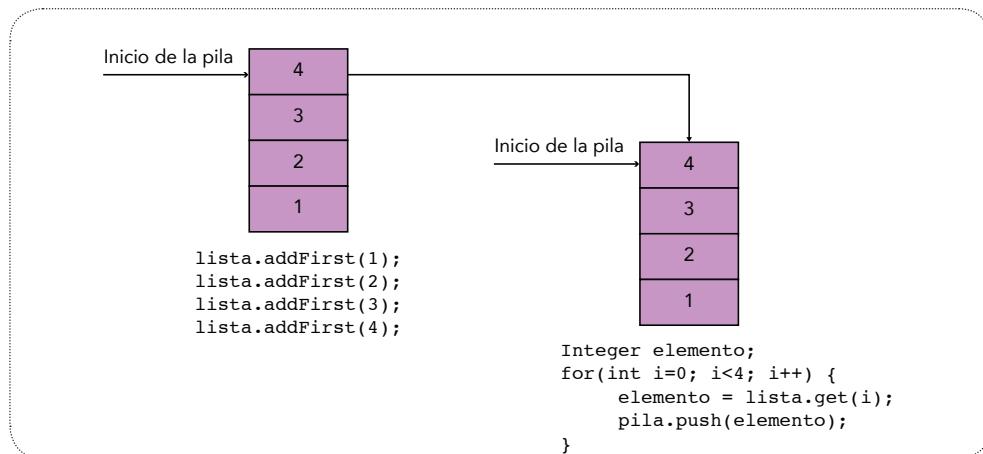


Figura IV-5. Estructura de la lista y de la pila en la memoria

Ejercicio 2. Modificar el programa del ejercicio anterior para que pida elementos de la lista sin tener que saber el tamaño exacto de la lista. Además, que despliegue todos los elementos que saca de la pila mientras ésta todavía contenga elementos, sin tener que saber el tamaño exacto de la pila.

Solución:

```

import java.util.*;

public class PilasMain {

    public static void main( String[] args ) {

        Stack<Integer> pila = new Stack<Integer>();
        LinkedList<Integer> lista = new LinkedList<Integer>();

        lista.addFirst( 1 );
        lista.addFirst( 2 );
        lista.addFirst( 3 );
        lista.addFirst( 4 );

        Integer elemento;
        for ( int i = 0; i < lista.size(); i++ ) {
            elemento = lista.get( i );
            pila.push( elemento );
        }

        System.out.println( "Los elementos que hay en la pila son:" );
        while( !pila.isEmpty() ) {
            elemento = pila.pop();
            System.out.println( "se sacó el elemento " + elemento +
        
```

Con el método `size()` ya no es necesario proporcionar el número de elementos de la lista.

El método `isEmpty()` regresa `true` cuando la pila está vacía.

```
        " de la pila ");  
    }  
}  
}
```

Cuando se termina de ejecutar el programa anterior no quedan elementos en la pila, ya que todos se fueron sacando con el método `pop()`. Para obtener un elemento de la pila sin sacarlo de ésta, utilizamos el método `get(int i)`, en donde `i` es el índice del elemento que queremos obtener.

Ejercicio 3. Hacer un programa que añada cuatro números a una lista ligada en la primera posición, después, que añada cada elemento de la lista a una pila. Además, que posteriormente muestre los elementos de la pila y la posición en la que se encuentra cada uno de ellos dentro de ésta.

Solución:

```
import java.util.*;  
  
public class PilasMain {  
  
    public static void main( String[] args ) {  
  
        Stack<Integer> pila = new Stack<Integer>();  
        LinkedList<Integer> lista = new LinkedList<Integer>();  
  
        lista.addFirst( 5 );  
        lista.addFirst( 6 );  
        lista.addFirst( 7 );  
        lista.addFirst( 8 );  
  
        Integer elemento;  
        for ( int i = 0; i < 4; i++ ) {  
            elemento = lista.get( i );  
            pila.push( elemento );  
        }  
  
        System.out.println( "Los elementos que hay en la pila son:" );  
  
        for ( int i = 0; i < 4; i++ )  
            System.out.println( "el número " + pila.get( i ) +  
                               " está en la posición " +  
                               pila.search( pila.get( i ) ) );  
  
    }  
}
```

El método `search()` regresa la posición en la que se encuentra un elemento a buscar. Si el elemento no está en la pila, regresa un `-1`.

El método **search()** regresa la posición en la que se encuentra un elemento a buscar. Si el elemento no está en la pila, regresa un **-1**.

Ejercicio 6. Determina, sin ver la solución, cuál será la salida del programa anterior.

Solución:

Los elementos que hay en la pila son:
 el número 5 está en la posición 1
 el número 6 está en la posición 2
 el número 7 está en la posición 3
 el número 8 está en la posición 4

En la figura IV-6 se muestra un esquema de lo que sucede en memoria. Nótese que la primera posición se encuentra al principio de la pila y que el índice i es diferente de la posición.

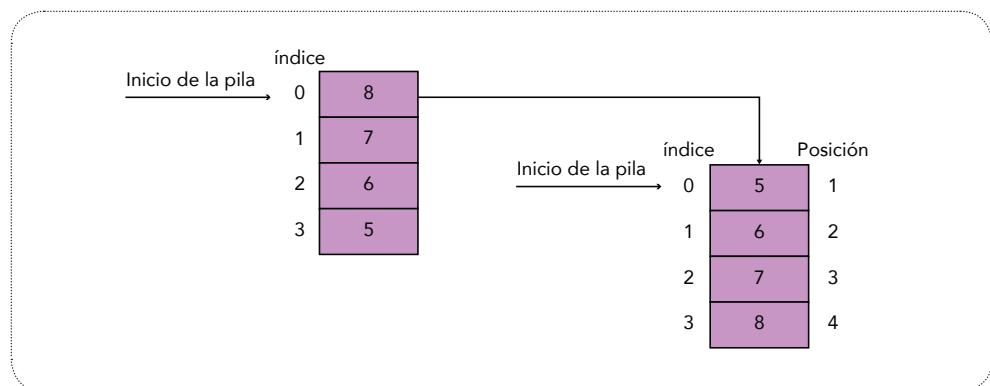


Figura IV-6 Posición de los elementos dentro de la pila

Concepto de fila (cola)

En una *fila* o *cola* el primer elemento que se saca de la lista es el primero que entró. A una cola también se le conoce como lista First In First Out (FIFO), es decir, el primero en entrar es el primero en salir. Una fila o cola en la programación funciona como una fila del banco o del supermercado.

Existen dos operaciones básicas para trabajar con filas, que son **formar()** y **despachar()**. La operación **formar()** se encarga de poner al final de la fila un elemento nuevo. El final de la fila apuntará ahora hacia el nuevo elemento que se formó. Podemos utilizar la clase **LinkedList** para trabajar con filas (colas). La operación **formar** se efectúa con el método **addLast()** de esta clase, el cual agrega el elemento al final de la fila, como se muestra en la figura IV-7.

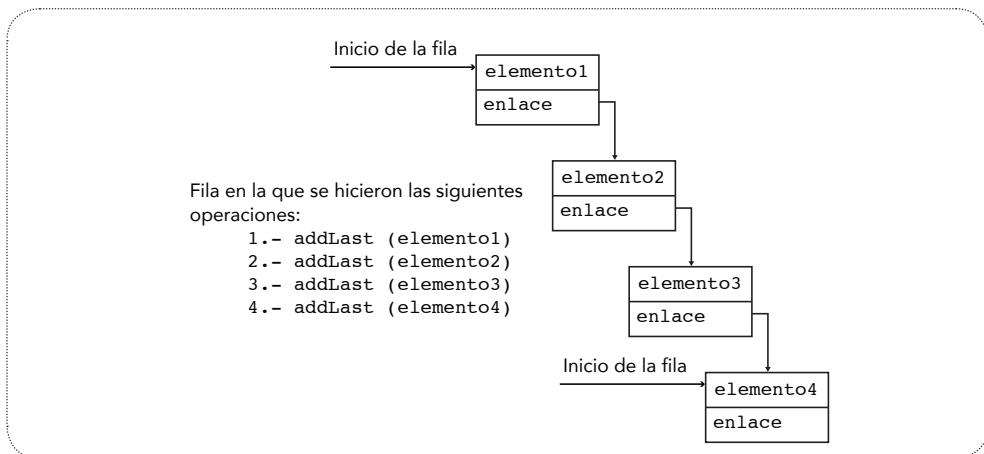


Figura IV-7. Implementación de una fila (cola)

La otra operación básica cuando se trabaja con filas es la función `despachar()`, que saca de la fila al primer elemento. En la figura IV-8 se ilustra la operación `despachar()`, que entrega el primer elemento y así el inicio apunta al siguiente elemento de la fila.

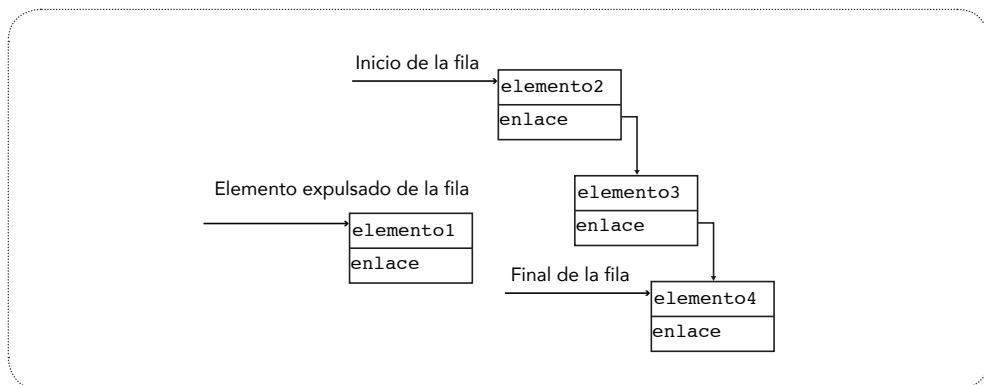


Figura IV-8. La operación `despachar()` expulsa al primer elemento de la fila

La operación `despachar()` se puede efectuar con el método `removeFirst()` de la clase `LinkedList`, el cual recupera y elimina de la fila el primer elemento. Si la fila está vacía, lanza la excepción `NoSuchElementException`.

Ejemplos de filas (colas)

Ejemplo 1. El siguiente programa inserta 4 enteros a una fila, posteriormente saca cada uno de los elementos de la lista y los guarda en una fila. Finalmente, saca los elementos de la fila y los va imprimiendo:

```

import java.util.LinkedList;

public class FilaMain {

    public static void main( String[] args ) {

        LinkedList<Integer> fila = new LinkedList<Integer>();
        Integer elemento;

        fila.addLast( 1 );
        fila.addLast( 2 );
        fila.addLast( 3 );
        fila.addLast( 4 );

        while ( !fila.isEmpty() ) {
            elemento = fila.removeFirst();
            System.out.println( "Se atendio al elemento: " + elemento
                + " de la fila");
        }

        System.out.println( "La fila esta vacia" );
    }
}

```

El método `isEmpty()` regresa `true` cuando ya no hay elementos en la lista.

Ejercicio 7. Determina, sin ver la solución, cuál será la salida del programa anterior.

Solución:

```

Se atendió al elemento: 1 de la fila
Se atendió al elemento: 2 de la fila
Se atendió al elemento: 3 de la fila
Se atendió al elemento: 4 de la fila
La fila está vacía

```

Ejemplo 2. Otra manera de implementar lo mismo que en el ejemplo anterior es utilizando el método `pollFirst()`, el cual recupera y elimina de la fila el primer elemento. Además, si la fila está vacía regresa `null`.

Ejemplo 3. Ahora haremos una fila y desplegaremos sus elementos sin eliminarlos de ésta. Mediante un menú, el usuario puede agregar elementos a la fila o desplegar los elementos ya existentes:

```

import java.util.LinkedList;
import Utererias.LectorDeTeclado;

public class FilaMain {

```

```
public static void main( String[] args ) {  
  
    LinkedList<String> fila = new LinkedList<String>();  
    String nombre;  
    int opcion;  
  
    while ( true ) {  
        do {  
            System.out.println( "elige una opcion\n" +  
                "1: Agregar un nombre\n" +  
                "2: Desplegar los nombres\n"+  
                "3: Salir" );  
            opcion = LectorDeTeclado.capturarEntero( "Opcion?" );  
        } while ( ( opcion < 1 ) || ( opcion > 3 ) );  
  
        switch ( opcion ) {  
            case 1: nombre = LectorDeTeclado.capturarLinea(  
                "Cuál es el nombre?" );  
                fila.addLast( nombre );  
                break;  
            case 2: for ( int i = 0; i < fila.size(); i++ )  
                System.out.println( "nombre " + (i + 1)  
                    + " es: " + fila.get( i ) );  
                break;  
            case 3: System.out.println( "Adios!" );  
                return;  
        }  
    }  
}
```

Ejercicio 8. Ejecuta el programa anterior y construye una lista de 5 nombres y desplíégala en pantalla.

Prácticas de laboratorio

Robot

Lectura de un escenario

En esta práctica se debe desarrollar la clase `LectorEscenario` que permita la lectura de un escenario desde un archivo de texto. Este archivo debe contener la información del escenario con el formato siguiente:

```
m n  
xr yr dr  
xo yo  
xi yi
```

en donde **m** y **n** son, respectivamente, las longitudes vertical y horizontal, **xr** y **yr** son las coordenadas del robot, **dr** es la dirección del robot, **xo** y **yo** son las coordenadas del objetivo y el resto de las coordenadas **xi** y **yi** corresponden a la ubicación de los obstáculos.

Esta clase debe contener el método **leer(String NombreArchivo)**, el cual recibe como **String** el nombre del archivo de texto que contiene la información del escenario. Este método debe devolver un objeto **Escenario**, cuya clase se describe a continuación.

La clase **Escenario** se usa para representar un espacio bidimensional, mediante la clase **Espacio**, y un robot de tipo **SuperRobot**. La clase debe contener sus respectivos métodos *getters* y *setters*.

Escribe una clase principal para hacer uso de tu clase **LectorEscenario**.

Tienda virtual

Lectura y escritura de un catálogo

En esta práctica se considera el desarrollo de las dos clases **LectorCatalogo** y **EscritorCatalogo** para leer y escribir, respectivamente, un catálogo de un archivo binario, utilizando el concepto de serialización.

La clase **LectorCatalogo** tiene el método **leer(String NombreArchivo)**, el cual recibe como **String** el nombre del archivo binario que contiene la información del catálogo. Este método deserializa el **Catalogo** contenido en el archivo y devuelve un objeto **Catalogo**, el cual contiene toda la información del archivo binario.

La clase **EscritorCatalogo** contiene el método **escribir(Catalogo catalogo, String NombreArchivo)**, que recibe el **Catalogo** a serializar y el nombre del archivo como **String** en donde se almacenará.

Escribe una clase principal para hacer uso de las clases **LectorCatalogo** y **EscrítorCatalogo**.

Cuestionario

Contesta las siguientes preguntas:

1. ¿En qué consiste el mecanismo de las excepciones?
2. ¿Para qué se usan las excepciones?
3. ¿Cuál es la diferencia entre usar la clase `Scanner` y hacer una clase propia para leer datos del teclado?
4. ¿Qué es necesario tomar en cuenta para hacer un arreglo con objetos?
5. ¿Cuál es la diferencia entre una pila y una fila (o cola)? ¿Qué clases se utilizan para trabajar con cada una de estas estructuras?

Polimorfismo y herencia múltiple

Objetivos

- Comprender el concepto de polimorfismo.
- Comprender el concepto de herencia múltiple.
- Aplicar las interfaces de Java para implementar el polimorfismo y la herencia múltiple.

Polimorfismo

El polimorfismo, en la POO, es la propiedad que tienen los objetos de las subclases de mostrarse como de la superclase. Por ejemplo, si tenemos dos clases X y Z que tienen el mismo ancestro W, entonces podemos usar una variable de clase W que pueda tomar el valor de un objeto de clase X o también el valor de un objeto de clase Z, como se muestra en la figura V 1.

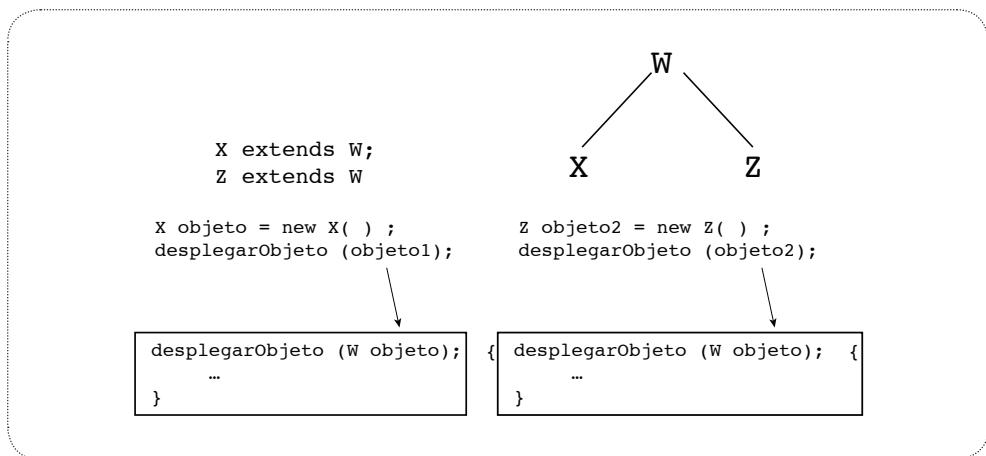


Figura V 1. Polimorfismo: W puede verse como un objeto X o un objeto Z

Ejemplo. Sea W la clase Persona, X la clase Alumno y Z la clase Profesor. Recordemos estas clases.

```
class Persona {  
    private String nombre;  
    private Fecha fechaNacimiento;
```

```
public Persona( String n, Fecha fn ) {
    nombre = n;
    fechaNacimiento = new Fecha( fn );
}

public void toString() {
    return nombre + " nacido el:" + fechaNacimiento;
}
}

public class Alumno extends Persona {
    private String matricula;

    public Alumno( String n, Fecha f, String m ) {
        super( n, f );
        matricula = m;
    }
    public String toString() {
        return super.toString() + " mat:" + matricula;
    }
}

public class Profesor extends Persona {

    private Integer claveEmpleado;

    public Profesor( String n, Fecha f, Integer c ) {
        super( n, f );
        claveEmpleado = c;
    }

    public String toString() {
        return super.toString() + " clave:" + claveEmpleado;
    }
}
```

La relación de herencia entre estas clases se ilustra en la figura V-2.

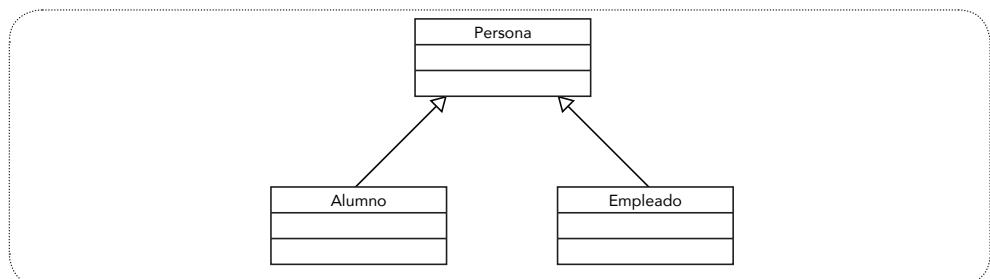


Figura V-2. Relación de herencia entre las clases Persona, Alumno y Profesor

A continuación haremos un programa con una lista de la clase **Persona** a la que llamaremos **listaPersonas**. Se dice que **listaPersonas** es una *lista polimórfica* porque puede recibir elementos con la forma de **Alumno** y también con la forma de **Profesor**.

```
import java.util.LinkedList;
import Utlcerias.LectorDeTeclado;

public class ListaPolimorficaMain {

    static LinkedList <Persona> listaPersonas =
        new LinkedList<Persona>();

    public static void main( String[] args ) {
        Alumno alumno;
        Profesor profesor;
        int opcion;

        while( true ) {
            do {
                System.out.print( "1... Capturar un alumno\n" +
                    "2... Capturar un profesor\n" +
                    "3... Salir\n" );
                opcion = LectorDeTeclado.capturarEntero(
                    "elige una opcion" );
            } while( ( opcion < 1 ) || ( opcion > 3 ) );

            switch( opcion ) {
                case 1: alumno = capturarAlumno();
                    agregarPersona( alumno );
                    break;

                case 2: profesor = capturarProfesor();
                    agregarPersona( profesor );
                    break;

                case 3: System.out.println(
                    "Las personas que ingresaste son:" );
                    desplegarLista( listaPersonas );
                    System.out.println( "adios!" );
                    return;
            }
        }
    }
}
```

Se agrega un objeto de clase
Alumno y también uno de
clase Profesor.

Los métodos para capturar un alumno y un profesor son los siguientes.

```
public static Alumno capturarAlumno() {
    Alumno dato;
    dato = new Alumno( LectorDeTeclado.capturarLinea( "nombre?" ),
        new Fecha(
            LectorDeTeclado.capturarEntero( "año nacimiento?" ),
            LectorDeTeclado. capturarEntero( "mes?" ),
            LectorDeTeclado. capturarEntero( "dia?" )
        ),
        LectorDeTeclado. capturarLinea( "matricula?" )
    );
    return dato;
}

public static Profesor capturarProfesor() {
    Profesor dato;
    dato = new Profesor(
        LectorDeTeclado.capturarLinea(
            "nombre del profesor?" ),
        new Fecha(
            LectorDeTeclado.capturarLinea(
                "año de nacimiento?" ),
            LectorDeTeclado.capturarLinea( "mes?" ),
            LectorDeTeclado.capturarLinea( "dia?" )
        ),
        LectorDeTeclado.capturarLinea(
            "clave de empleado?" )
    );
    return dato;
}
```

Y los métodos para agregar una persona y para desplegar la lista polimórfica son, respectivamente:

```
public static void agregarPersona( Persona x ) {
    listaPersonas.addLast( x );
}

public static void desplegarLista( LinkedList<Persona> l ){
    for ( int i = 0; i < l.size(); i++ )
        System.out.println( l.get( i ) );
}
```

Enseguida hay un ejemplo de uso de este programa en donde se capturan los datos de un **Alumno** y de un **Profesor** para luego desplegar la lista con ambos.

```
1... Capturar un alumno
2... Capturar un profesor
3... Salir
elige una opcion
1
nombre del alumno?
Pedro Perez
año de nacimiento?
1987
mes?
12
dia?
28
matricula?
2098765
1... Capturar un alumno
2... Capturar un profesor
3... Salir
elige una opcion
2
nombre del profesor?
Elmer Homero Petatero
año de nacimiento?
1961
mes?
09
dia?
11
clave de empleado?
44567
1... Capturar un alumno
2... Capturar un profesor
3... Salir
elige una opción
3

Las personas que ingresaste son:

Pedro Perez nacido el 28 de Dic de 1987 mat: 2098765
Elmer Homero Petatero nacido el 11 de Sep de 1961 clave: 44567
adios!
```

Como se observa, se despliega una lista de objetos **Persona** que pueden ser objetos tanto de clase **Alumno** como de clase **Profesor**.

Interfaces

El usuario de un objeto se comunica con el objeto mediante la interfaz del objeto. Una interfaz es la *vista externa* de una “máquina” la cual oculta la estructura y los detalles de su comportamiento. Todo elemento externo a la máquina interactúa con ésta a través de su interfaz y no lo puede hacer de otra manera. Por ejemplo, al solicitar a una calculadora la raíz cuadrada de 24, el resultado aparece en la interfaz sin importar cómo se obtuvo. Además el usuario no puede intervenir en el procedimiento para obtener el resultado. En el caso particular de la POO, los usuarios son las clases que usan un determinado objeto.

La interfaz de una clase está conformada por los métodos y atributos que el usuario emplea para lograr que un objeto se comporte de cierta forma. Así, por ejemplo, la interfaz de una lista está formada por los siguientes métodos:

- `agregarAlFinal`
- `agregarAlInicio`
- `tomarElUltimo`
- `tomarElPrimero`

Otro ejemplo son los métodos de la interfaz de una pila:

- `push` (para agregar un elemento en ella)
- `pop` (para sacar de ella un elemento)

Finalmente, los métodos de la interfaz de una cola:

- La operación formar
- La operación despachar

En el capítulo anterior escribimos la clase `LectorDeTeclado`, la cual nos permite leer datos desde el teclado. En este caso, la interfaz de la clase `LectorDeTeclado` está formada por los siguientes métodos:

- `capturarEntero()`
- `capturarDoble()`
- `capturarLinea()`
- `capturarArrDoble()`

En Java es posible definir una interfaz sin asociarla a ninguna clase y se le da un nombre. Se define cuáles métodos y cuáles atributos (`static final`) la conforman y, al definir una clase, se puede especificar cuál o cuáles interfaces serán parte de esa clase. La sintaxis para definir una interfaz en java es la siguiente:

```
public interface <NombreInterfaz> {  
  
    public static final <tipo> <nombreAtributoInterfaz> = <valor>;  
    ... otros atributos  
  
    public <tipo> <NombreMetodoInterfaz>(<parámetros> );  
    ... otros métodos sin cuerpo  
}
```

Una interfaz, para que sea útil, debe ser *implementada* por alguna clase, es decir, en la interfaz se definen ciertos atributos y métodos, y en la clase se implementan los métodos de la interfaz. De ser necesario, en la clase que implementa a la interfaz se pueden declarar atributos y métodos adicionales.

La sintaxis para definir la clase que implementa una interfaz en Java es la siguiente:

```
class NombreClase implements <NombreInterfaz> {  
  
    public <tipo> <NombreMetodoInterfaz>(<parametros> ) {  
        ... cuerpo del método  
    }  
  
    ... declaración con cuerpo de cada metodo en la interfaz  
  
    ... declaraciones adicionales propias de esta clase  
}
```

En la figura V-3 se aprecia la notación que usa UML para indicar cuando una clase A implementa a una Interfaz1.

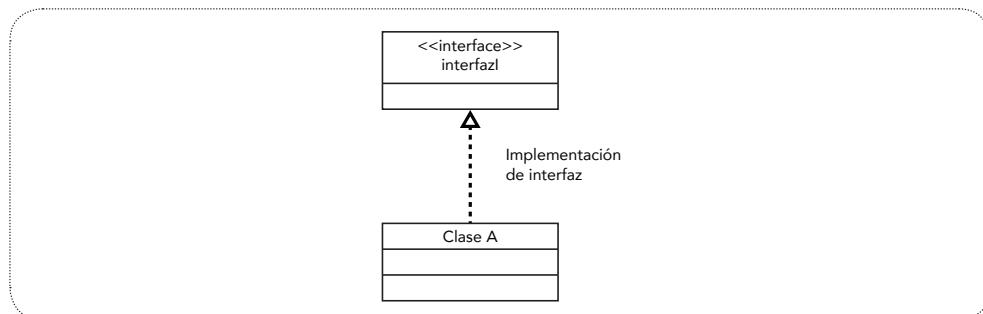


Figura V-3. Notación UML para interfaces

Propiedades de una interfaz

La separación entre especificación e implementación es fundamental en la ingeniería de sistemas software. Las interfaces son útiles porque en la interfaz se especifican las características (métodos y atributos) requeridas y en las clases se implementan estas características. Un beneficio fundamental de las interfaces es que podemos usar implementaciones diferentes (clases) para una misma especificación (interfaz), es decir, se pueden construir clases que implementen una misma interfaz pero con estrategias diferentes. Una interfaz tiene las siguientes propiedades:

- Permite al programador establecer la forma que tendrá una clase, es decir, el nombre de los métodos, la lista de argumentos de los métodos y sus tipos de retorno.
- También puede contener atributos. Es importante mencionar que estos deben ser `static` y `final`. La palabra `static` significa que siempre ocupa la misma posición en memoria y por lo tanto todos los objetos instanciados usarán la misma variable. La palabra `final` significa que no puede modificarse.
- Proporciona sólo una definición, jamás una implementación.
- Se utiliza para establecer un protocolo entre clases, es decir, establecer qué métodos deben tener las clases que implementen la interfaz.
- Se implementa en un archivo `*.java` y sigue las mismas reglas de visibilidad y reagrupamiento en paquetes que las clases.

Interfaces múltiples

Una misma clase puede implementar varias interfaces. Cuando se utilizan interfaces diferentes en un mismo objeto significa que cada una de ellas tiene capacidades de interacción diferentes, no correlacionadas. Por ejemplo, supongamos que definimos la interfaz `IPrintable`, que es una interfaz para objetos que despliegan su contenido en la pantalla mediante el método `print()`.

```
public interface IPrintable {  
    void print();  
}
```

Además, tenemos la interfaz `ICounter` para objetos que implementan un contador, que incluyen los métodos que se muestran a continuación:

```
public interface ICounter {  
    void incrementar();  
    void setValue( int v );  
    int getValue();  
}
```

Entonces, podemos definir un contador “imprimible” con un objeto que implementa tanto la interfaz `ICounter` como la interfaz `IPrintable`, como se ilustra en el siguiente ejemplo:

```
public class PrintableCounter implements ICounter, IPrintable {  
    private int cont;  
  
    public PrintableCounter() {  
        cont = 0;  
    }  
  
    public PrintableCounter( int v ) {  
        cont = v;  
    }  
  
    public void setValue( int v ) {  
        cont = v;  
    }  
  
    public void incrementar() {  
        cont++;  
    }  
  
    public int getValue() {  
        return cont;  
    }  
  
    public void print() {  
        System.out.println( "Contador " + this  
                            + " Valor actual " + cont );  
    }  
}
```

En la figura V-4 se ilustra con UML que la clase `PrintableCounter` implementa a dos interfaces: `ICounter` e `IPrintable`.

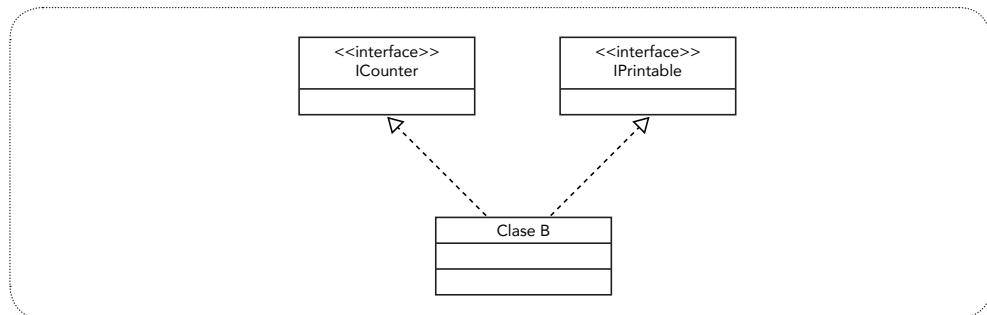


Figura V-4. Diagrama UML para la implementación de dos interfaces

Polimorfismo por interfaces

Supongamos que tenemos la siguiente interfaz Reproductor:

```
public interface Reproductor {  
    public void playPause();  
    public void fastForward();  
    public void fastRewind();  
    public void stop();  
}
```

Y las siguientes clases que implementan la interfaz Reproductor:

```
public class Videocamara implements Reproductor {  
    ...  
}  
  
public class IPod implements Reproductor {  
    ...  
}  
  
public class Contestadora implements Reproductor {  
    ...  
}
```

Todas estas clases deberán implementar los métodos definidos en la interfaz Reproductor, además de los métodos propios de cada una.

Los objetos de las clases Videocamara, IPod y Contestadora pueden ser tratados indistintamente si son vistos como “Reproductores”. Es posible, por ejemplo, declarar una variable de tipo Reproductor y asignarle un objeto de cualquiera de estas 3 clases, como en el siguiente ejemplo.

```
Reproductor aparato1;  
Reproductor aparato2;  
Reproductor aparato3;  
aparato1 = new Videocamara(...);  
aparato2 = new IPod(...);  
aparato3 = new Contestadora(...);
```

Desde el objeto aparato1 sólo pueden invocarse los métodos de Reproductor aunque el objeto, por ser de la clase Videocamara tenga otros métodos. En este caso sólo es posible invocar los métodos del aparato1 que forman parte de la interfaz Reproductor. Lo mismo sucede para los objetos aparato2 y aparato3.

De igual manera, puede haber algún método que reciba como parámetro un objeto descrito por alguna interfaz. Por ejemplo, le encargamos al DJ que use el **aparato** que le indiquemos. Él ya sabe usarlo, siempre y cuando cumpla con la interfaz **Reproductor**.

```
public void DJ( Reproductor aparato ) {
    aparato.play();
    aparato.fastForward();
    aparato.stop()
}
```

El parámetro **aparato** que se recibe en el método **DJ** puede ser **aparato1** (la videocámara), **aparato2** (el iPod) o **aparato3** (la contestadora). En este caso, el parámetro **aparato** es polimórfico ya que puede tener cualquiera de estas tres "formas" o cualquiera otra que implemente la interfaz **Reproductor**.

En la figura V-5 las clases **Videocamara**, **IPod** y **Contestadora** implementan a la misma interfaz: **Reproductor**.

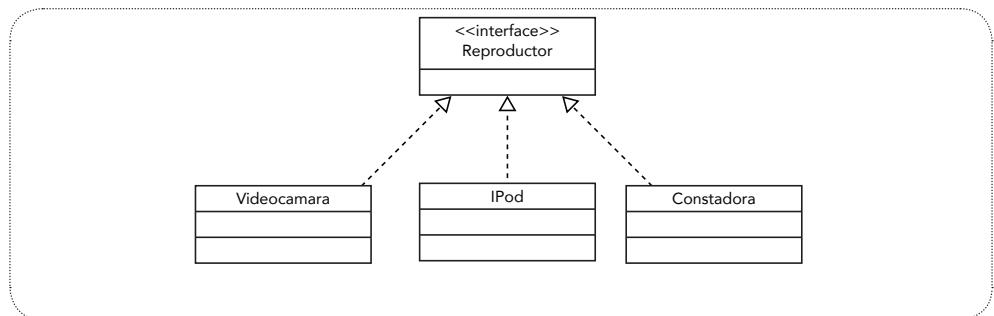


Figura V-5. Tres clases que implementan a la misma interfaz

Herencia múltiple

Con la herencia múltiple una subclase puede heredar propiedades de más de una superclase.

La herencia múltiple por interfaces

La herencia múltiple en Java no existe, pero puede implementarse (en cierta medida) mediante interfaces. En la figura V-6 presentamos un ejemplo en el que todas las clases descienden de la clase padre **Intercomunicador**. Hay dos clasificaciones:

- OneWay
- TwoWay

que son dos clases que sirven para distinguir cuando la comunicación del intercomunicador es de una vía (sólo se habla o sólo se escucha) o de dos vías (se puede hablar y escuchar al mismo tiempo). Además,

- Fixed
- Mobile

son dos interfaces que sirven para distinguir un aparato que está fijo, de uno móvil. En este ejemplo existen las siguientes relaciones de herencia múltiple:

- El interfono es un intercomunicador de una sola vía que además es fijo.
- El walkie talkie es un intercomunicador de una sola vía que, además, es móvil.
- El teléfono de casa (HomePhone) es de dos vías y además es fijo.
- El teléfono celular es de dos vías y además es móvil.

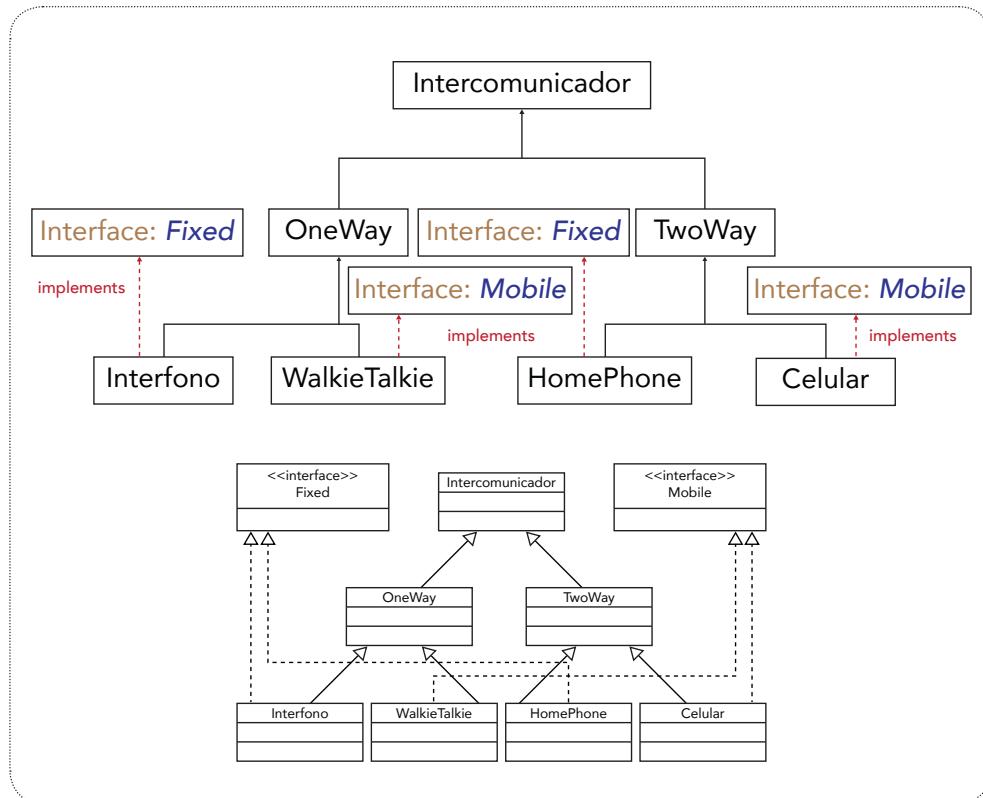


Figura V-6. Ejemplo de herencia múltiple implementada con interfaces y clases

Como se observa en la figura V-6, la herencia múltiple se establece haciendo que una clase herede de la clase padre con la relación “*es un*” y que además se “implemente una” interfaz, de la siguiente manera:

- El `Interfono` `extends OneWay` y también `implements Fixed`.
- El `WalkieTalkie` `extends OneWay` y también `implements Mobile`.
- `HomePhone` `extends TwoWay` y también `implements Fixed`.
- `Celular` `extends TwoWay` y también `implements Mobile`.

La clase `Intercomunicador` describe a un aparato que sirve para enviar y recibir mensajes, es una clase abstracta que tiene como atributo un número de serie (identidad), un método para enviar mensaje y el método `toString()`.

```
public abstract class Intercomunicador {
    private String serialNum;

    public Intercomunicador( String n ) {
        serialNum = new String( n );
    }

    public abstract Boolean sendMessage( String m );

    public String toString() {
        return serialNum;
    }
}
```

Cuando un método es abstracto, las clases hijas deben implementarlo.

`OneWay` es un `Intercomunicador` de *una vía*, y funciona de la siguiente forma:

- Para enviar mensajes se debe oprimir el botón maestro.
- Para escuchar mensajes se debe liberar el botón maestro.
- Todos los aparatos escuchan los mensajes enviados.

`TwoWay` es un `Intercomunicador` de *dos vías* que funciona de la siguiente manera:

- Se debe establecer una llamada desde un aparato hasta otro para poder iniciar la comunicación.
- Se pueden enviar y recibir mensajes al mismo tiempo.
- Los mensajes sólo son escuchados por el aparato con el que se estableció la llamada.

Por otra parte, tenemos que, `Fixed` es una interfaz que implementa un aparato que se fija en algún lugar, por ejemplo, cocina, baño, etc., por lo tanto la interfaz tiene:

- Un atributo de localización.
- Un método para poner la localización.

El código para la interfaz **Fixed** es el siguiente:

```
public interface Fixed {
    public void setAddress( String addr );
}
```

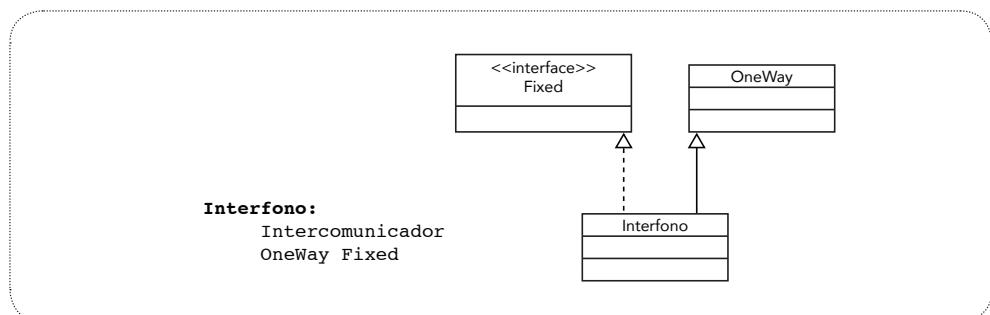
Mobile es una interfaz que implementa un aparato que posee un botón de encendido/apagado. Cuando el botón está en apagado ninguna de sus funciones está disponible, por lo tanto la interfaz tiene:

- Un método que se comporta como botón de encendido/apagado, **onOff()**.
- Un método para saber si el aparato está encendido, **isOn()**.

El código para la interfaz **Mobile** es el siguiente:

```
public interface Mobile {
    public Boolean onOff();
    public Boolean isOn();
}
```

En la figura V-7 se observa que cada aparato hereda dos características, una mediante la implementación de una interfaz (**Fixed** ó **Mobile**) y la otra característica la hereda de una clase padre (**OneWay** ó **TwoWay**). Como las clases **OneWay** y **TwoWay** son hijas de la clase **Intercomunicador**, entonces cada aparato es un **Intercomunicador**.



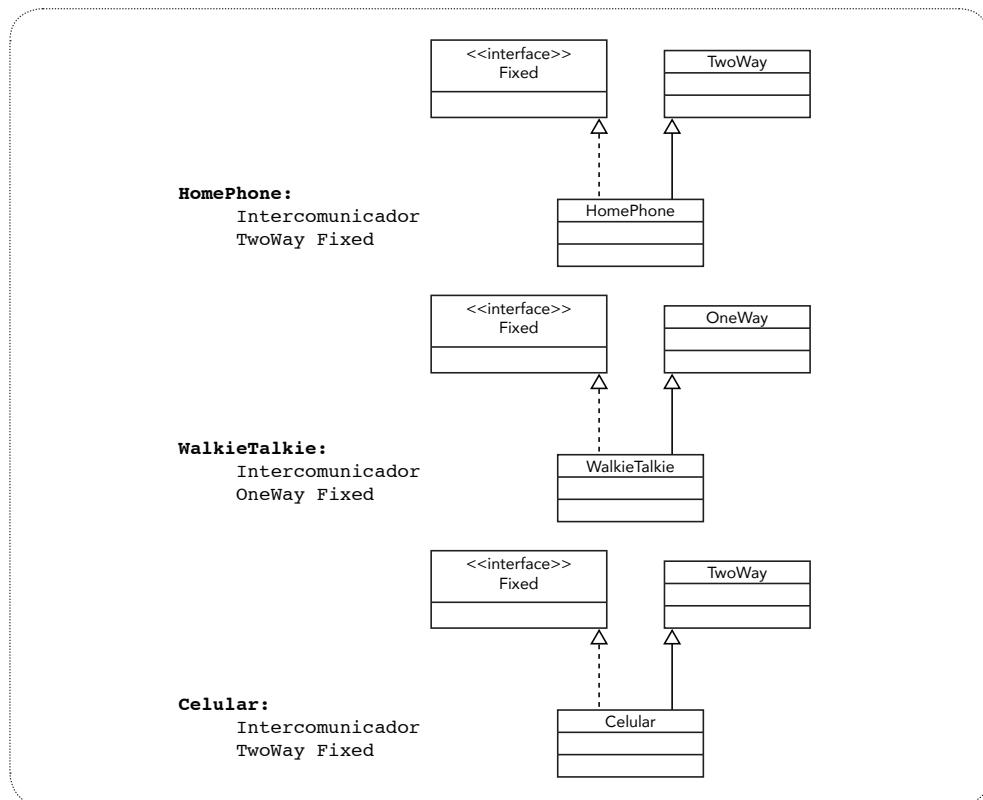


Figura V-7. Cada aparato hereda de una clase e implementa una interfaz

A continuación presentamos el código parcial de la clase `OneWay`. El código completo está en el apéndice 1.

```

import java.util.*;

public abstract class OneWay extends Intercomunicador {
    static final Integer LISTEN = 0;
    static final Integer TALK = 1;
    private Integer state;
    protected LinkedList<OneWay> listeners;

    public OneWay( String serialNum ) {}

    public void push() {}

    public void release() {}

    protected Boolean addListener( OneWay listener ) {}

    protected void removeListener( OneWay x ) {}
  
```

```
public Boolean sendMessage( String m ) {}

protected Boolean receiveMessage( String m ) {}

public String toString() {}

}
```

La clase `Interfono` es un intercomunicador fijo de una sola vía, por esto hereda de `OneWay` e implementa `Fixed`. Las clases que implementan `Fixed` deben codificar el método `setAddress()`. El código completo de esta clase está en el apéndice 1. He aquí el código parcial:

```
public class Interfono extends OneWay implements Fixed {
    private String address;

    public Interfono( String serialNum ) {}

    public void connectTo( Interfono x ) {}

    public void setAddress( String addr ) {}

    public String toString() {}

}
```

La clase `WalkieTalkie` es un intercomunicador móvil de una sola vía, por esto hereda de `OneWay` e implementa `Mobile`. Las clases que implementan la interfaz `Mobile` deben codificar los métodos `onOff()` e `isOn()`. En `WalkieTalkie` se redefinen algunos métodos de la clase padre `OneWay` para verificar que el aparato esté prendido antes de usar el método. El código completo está en el apéndice 1. Aquí está el código parcial:

```
public class WalkieTalkie extends OneWay implements Mobile {
    private Boolean on;

    public WalkieTalkie( String serialNum ) {}

    public void push() {}

    public void release() {}

    public Boolean sendMessage( String m ) {}

    protected Boolean receiveMessage( String m ) {}

    public Boolean onOff() {}

}
```

```

public Boolean isOn() {}

public String toString() {}

}

```

Para implementar la clase `TwoWay`, añadimos el concepto de *central telefónica* (`Exchange`). Cada aparato se asocia con una *central* (`Exchange`) y ésta tiene una lista de los aparatos que pueden comunicarse entre sí, como se ilustra en la figura V-8.

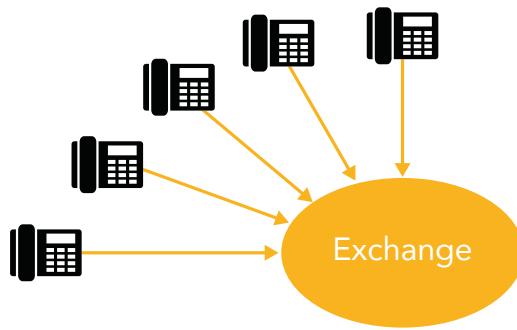


Figura V-8. Una central telefónica (`Exchange`) tiene asociados varios teléfonos que se comunican entre sí

La clase `Exchange` es, en este ejemplo, hija de una lista ligada de objetos `TwoWay`. Adicionalmente a los métodos de una lista ligada, `Exchange` contiene el método `find()` que recibe como parámetro un número telefónico y regresa una referencia a un objeto `TwoWay` si encuentra uno con el número dado dentro de la lista, en caso contrario regresa `null`:

```

import java.util.*;

public class Exchange extends LinkedList<TwoWay> {

    public TwoWay find( String num ) {
        for ( int i = 0; i < size(); i++ ) {
            if ( get(i).number().equalsIgnoreCase( num ) ) {
                return get( i );
            }
        }
        return null;
    }
}

```

El código de la clase `TwoWay` contiene tres indicadores de estado: cuando el aparato está desocupado (`IDLE` = 0), cuando está ocupado (`BUSY` = 1) y cuando está timbrando (`RINGING` = 2). El método `dial()` se encarga de solicitar la conexión con

un número dado, tomando en cuenta que este número existe en la central (Exchange) e intentando hacerlo timbrar. Si lo logra, el estado del aparato cambia a BUSY. El método `endCall()` desconecta los teléfonos. Finalmente, los métodos `ring()` y `answerCall()`, contienen la lógica de timbrado y de contestar la llamada, respectivamente. Adicionalmente, `TwoWay` contiene métodos para enviar y recibir mensajes. El código completo está en el apéndice 1. Aquí está el código parcial:

```
public abstract class TwoWay extends Intercomunicador {  
    public static final Integer IDLE = 0;  
    public static final Integer BUSY = 1;  
    public static final Integer RINGING = 2;  
    private Integer state;  
    private String number;  
    private Exchange ownExchange;  
    private TwoWay other;  
  
    public TwoWay( String serialNum, String num, Exchange e ) {}  
  
    public Boolean dial( String num ) {}  
  
    public void endCall() {}  
  
    private void callEnded() {}  
  
    protected Boolean ring( TwoWay sender ) {}  
  
    public void answerCall( String m ) {}  
  
    public String number() {}  
  
    public Boolean sendMessage( String m ) {}  
  
    protected Boolean receiveMessage( String m ) {}  
  
    public String toString() {}  
}
```

La clase `HomePhone` representa a un teléfono de casa, por lo que la comunicación es de dos vías y es fijo, y por lo tanto hereda de `TwoWay` e implementa `Fixed`. Las clases que implementan `Fixed` deben codificar el método `setAddress()`. El código completo está en el apéndice 1. Aquí está el código parcial:

```
public class HomePhone extends TwoWay implements Fixed {  
    private String address;  
  
    public HomePhone( String serialNum, String num, Exchange e ) {}
```

```

public String toString() {}

public void setAddress( String addr ) {}

}

```

La clase **Celular**, que representa a un teléfono celular, establece también una comunicación de dos vías, pero es móvil, por lo que hereda de **TwoWay** e implementa **Mobile**. Las clases que implementan la interfaz **Mobile** deben codificar los métodos **onOff()** e **isOn()**. En la clase **Celular** se redefinen los métodos para llamar, timbrar y recibir mensajes, ya que éstos sólo funcionan si el celular está prendido. El código completo está en el apéndice 1. Aquí está el código parcial:

```

public class Celular extends TwoWay implements Mobile {
    private Boolean on;

    public Celular( String serialNum, String number, Exchange e ) {}

    public Boolean dial( String n ) {}

    public Boolean receiveMessage( String m ) {}

    public void ring() {}

    public Boolean onOff() {}

    public Boolean isOn() {}

    public String toString() {}
}

```

El siguiente programa hace uso de los cuatro dispositivos: Interfono, **WalkieTalkie**, **HomePhone** y **Celular**:

```

public class IntercomunicadorMain {

    public static void main(String[] args) {
        jugarConInterfonos();
        jugarConWalkieTalkies();
        jugarConHomePhones();
        jugarConCelulares();
    }

    private static void jugarConInterfonos() {
        Interfono[] interfonos = {
            new Interfono( "IF1111" ),
            new Interfono( "IF0111" ),
            new Interfono( "IF0011" ),

```

Instancia un arreglo de cuatro **Interfonos** proporcionando una dirección a cada uno.

```

        new Interfono( "IF0001" )
    };
    interfonos[0].setAddress( "Cocina" );
    interfonos[1].setAddress( "Baño" ); ← Asigna un lugar a cada Inter-
    interfonos[2].setAddress( "Estudio" );
    interfonos[3].setAddress( "Patio" );
    interfonos[0].connectTo( interfonos[1] );
    interfonos[0].connectTo( interfonos[2] );
    interfonos[0].connectTo( interfonos[3] );
    interfonos[1].connectTo( interfonos[2] );
    interfonos[1].connectTo( interfonos[3] );
    interfonos[2].connectTo( interfonos[3] );
    jugarConOneWay( interfonos[0],
                    interfonos[1],
                    interfonos[2],
                    interfonos[3] );
}

private static void jugarConHomePhones() {
    Exchange e = new Exchange();
    HomePhone[] HP = {
        new HomePhone( "HP1111", "5558991122", e ),
        new HomePhone( "HP0111", "5521902002", e ),
        new HomePhone( "HP0011", "5555626262", e ),
        new HomePhone( "HP0001", "5521188888", e )};
    HP[0].setAddress( "Ciruelos 24, Naucalpan" );
    HP[1].setAddress( "Oyameles 1, Pedregal" );
    HP[2].setAddress( "Grietas 33, Lomas Gacho" );
    HP[3].setAddress( "Rosa 24, Nativitas" );
    jugarConTwoWay( HP[0], HP[1], HP[2], HP[3] );
}

private static void jugarConWalkieTalkies() {
    WalkieTalkie[] walkieTalkies = {
        new WalkieTalkie( "WT1111" ),
        new WalkieTalkie( "WT1110" ),
        new WalkieTalkie( "WT1100" ),
        new WalkieTalkie( "WT1000" ) };
```

Instancia un arreglo de cuatro HomePhones proporcionando a cada uno una dirección, un número y una central a la que se conecta.

Instancia un arreglo de cuatro WalkieTalkies, proporcionando una dirección a cada uno.

Interconecta los WalkieTalkies entre sí.

```

}

private static void jugarConCelulares() {
    Exchange e = new Exchange();
    Celular[] C = {
        new Celular( "HP1111", "5558991122", e ),
        new Celular( "HP0111", "5521902002", e ),
        new Celular( "HP0011", "5555626262", e ),
        new Celular( "HP0001", "5521188888", e )
    };
    prenderMobiles( C );
    jugarConTwoWay( C[0], C[1], C[2], C[3] );
}

// prende los dispositivos móviles
private static void prenderMobiles( Mobile[] m ) {
    for ( int i = 0; i < m.length; i++ ) {
        if ( !m[i].isOn() ) {
            m[i].onOff();
        }
    }
}

// Ejemplo de comunicación entre dispositivos de una vía
private static void JugarConOneWay( OneWay a, OneWay b,
                                    OneWay c, OneWay d ) {
    a.push();
    a.sendMessage("Hola ¿ya terminaste de bañarte?"); a.release();
    b.push();
    b.sendMessage("Ya...¿porque?"); b.release();
    a.push();
    a.sendMessage("Ya esta tu cena..."); a.release();
    c.push();
    c.sendMessage("Yo también quiero. ¿qué hay?"); c.release();
    a.push();
    a.sendMessage("De todo lo de siempre"); a.release();
    c.push();
    c.sendMessage("¿me haces unas quechas porfa?"); c.release();
    a.push(); a.sendMessage("¿cuantas?"); a.release();
    c.push(); c.sendMessage("dos porfavor"); c.release();
    a.push(); a.sendMessage("OK"); a.release();
    c.push(); c.sendMessage("Gracias"); c.release();
}

// Ejemplo de comunicación entre dispositivos de dos vías
private static void JugarConTwoWay( TwoWay a, TwoWay b,
                                    TwoWay c, TwoWay d ) {
}

```

Instancia un arreglo de cuatro Celulares, proporcionando a cada uno una dirección, un número y una central a la que se conecta.

Recibe como parámetro un arreglo de objetos que implementan la interfaz Mobile (de Walkie-Talkies o de Celulares).

Las interfaces pueden usarse como clases para definir variables. En este caso, m es un arreglo de objetos que implementan la interfaz Mobile.

```
if( a.Dial("5521902002") ) {  
    b.answerCall("¿Bueno?");  
    a.sendMessage("¿Quien habla?");  
    b.sendMessage("Yo.");  
    a.sendMessage("¿Quien yo?");  
    b.sendMessage("Johnny Merote. ¿Con quien desea hablar?");  
    a.sendMessage("Con Juan Mero.");  
    b.sendMessage("Ah. No esta.");  
    a.sendMessage("Bueno. Hablo mas tarde. Gracias");  
    b.sendMessage("De nada.");  
    a.endCall();  
}  
  
if( a.dial("5521188888") ) {  
    d.answerCall("Hola, ¿quien habla?");  
    a.sendMessage("Rico Mac Pato.");  
    d.sendMessage("Hoooooola. Qué tal? Como estas?");  
    a.sendMessage("Pus, molestandote con un favorcito.");  
    d.sendMessage("Claro! Dime.");  
    a.sendMessage("Necesito que me prestes un dinerito.");  
    d.sendMessage("Tu? pero, como!?");  
    a.sendMessage("Es que necesito mas fortuna.");  
    d.sendMessage("Estas loco!");  
    a.sendMessage("Eso es un no?");  
    d.endCall();  
    a.sendMessage("oye, oye! uuuy!! Me colgo.");  
}  
}
```

Prácticas de laboratorio

Robot

Tus diseños de `Robot` y `SuperRobot` han tenido éxito, a tal grado, que muchos desarrolladores comenzaron a escribir aplicaciones basadas en tus clases. Entonces, es conveniente tener una interfaz para éstas y así los desarrolladores podrán utilizar de mejor forma tu diseño.

Escribe las interfaces `IntRobot` e `IntSuperRobot` y define a tus clases `Robot` y `SuperRobot` para que implementes las interfaces `IntRobot` e `IntSuperRobot` respectivamente. Además, escribe una clase principal que haga uso de las interfaces definidas.

Tienda virtual

Recordemos que la clase `Producto` tiene dos subclases, la clase `Libro` y la clase `Pelicula`. Con el fin de aplicar el concepto de polimorfismo, incluye en las tres clases el método `toString()`, de tal forma que:

- En la clase `Producto`, devuelva el `titulo` y el `precio` del producto.
- En la clase `Libro`, además del `titulo` y el `precio`, devuelva el `autor`.
- En la clase `Pelicula`, además del `titulo` y el `precio`, devuelva el `protagonista` y el `director`.

Escribe una clase principal que haga uso del concepto de polimorfismo recién añadido.

Cuestionario

Contesta las siguientes preguntas:

1. ¿Qué entiendes por polimorfismo?
2. ¿Qué relación debe existir entre las clases A, B y C para que los objetos de clase A puedan tomar la forma de los objetos de clase B y de los de clase C?
3. ¿Se deben codificar los métodos en una interfaz?
4. ¿Cuál es el mecanismo para codificar los métodos que se declaran en una interfaz?
5. ¿Cómo se implementa la herencia múltiple con Java?

Apéndice 1. Código

Clase OneWay

```
import java.util.*;  
  
public abstract class OneWay extends Intercomunicador {  
    static final Integer LISTEN = 0;  
    static final Integer TALK = 1;  
    private Integer state;  
    protected LinkedList<OneWay> listeners;  
  
    public OneWay( String serialNum ) {  
        super( serialNum );  
        state = LISTEN;  
        listeners = new LinkedList<OneWay>();  
    }  
  
    public void push() {  
        state = TALK;  
    }  
  
    public void release() {  
        state = LISTEN;  
    }  
  
    protected Boolean addListener( OneWay listener ) {  
        return listeners.add( listener );  
    }  
  
    protected void removeListener( OneWay x ) {  
        listeners.remove( x );  
    }  
  
    public Boolean sendMessage( String m ) {  
        Boolean success=true;  
        System.out.println( this + " envía: " + m );  
        for ( Integer i = 0; i < listeners.size(); i++ ) {  
            success = success && listeners.get( i ).receiveMessage( m );  
        }  
        return success;  
    }  
}
```

Cada objeto tiene una lista de aparatos que pueden escuchar sus mensajes.

```

protected Boolean receiveMessage( String m ) {
    if ( state != LISTEN ) {
        return false;
    }
    System.out.println( this + " recibio: " + m );
    return true;
}

public String toString() {
    String[] s = { "LISTEN", "TALK" };
    return super.toString() + " s:" + s[state];
}
}

```

Clase Interfono

```

public class Interfono extends OneWay implements Fixed {
    private String address;

    public Interfono( String serialNum ) {
        super( serialNum );
    }

    public void connectTo( Interfono x ) {
        super.addListener( x ); ←
        x.addListener( this );
    }

    public void setAddress( String addr ) {
        address = addr;
    }

    public String toString() {
        return super.toString() + " A: " + address; // toString()
                                            // de OneWay
    }
}

```

Los interfonos se conectan con otros interfonos.

Clase WalkieTalkie

```

public class WalkieTalkie extends OneWay implements Mobile {
    private Boolean on;

    public WalkieTalkie( String serialNum ) {
        super( serialNum );
        on = false;
    }
}

```

Lo único que tiene más allá de un OneWay es que implementa Mobile; es decir, debe revisar que esté prendido antes de hacer cualquier cosa.

```
public void push() {
    if( !on ) {
        return;
    }
    super.push();
}

public void release() {
    if( !on ) {
        return;
    }
    super.release();
}

public Boolean sendMessage( String m ) {
    if( !on ) {
        return false;
    }
    return super.sendMessage( m );
}

protected Boolean receiveMessage( String m ) {
    if( !on ) {
        return false;
    }
    return super.receiveMessage( m );
}

public Boolean onOff() {
    on = !on;
    return on;
}

public Boolean isOn() {
    return on;
}

public String toString() {
    return super.toString() + " on:" + on;
}
}
```

Clase TwoWay

```
public abstract class TwoWay extends Intercomunicador {
    public static final Integer IDLE = 0;
    public static final Integer BUSY = 1;
    public static final Integer RINGING = 2;
    private Integer state;
```

```
private String number;
private Exchange ownExchange;
private TwoWay other;

public TwoWay( String serialNum, String num, Exchange e ) {
    super( serialNum );
    state = IDLE;
    number = num;
    ownExchange = e; ← Cada aparato se asocia con una
    other = null; Exchange y esta Exchan-
    ownExchange.add( this ); ge tiene una lista de aparatos
} asociados a la misma.

public Boolean dial( String num ) {
    if ( state != IDLE ) {
        return false;
    }
    System.out.println( this + " llamando a " + num );
    other = ownExchange.find( num );
    if ( other == null ) {
        System.out.println( "numero:" + num + " no encontrado." );
        return false;
    }
    state = BUSY;
    if ( !other.ring( this ) ) {
        System.out.println( other + " no timbro." );
        state = IDLE;
        return false;
    }
    return true;
}

public void endCall() {
    other.callEnded();
    this.callEnded();
}

private void callEnded() {
    state = IDLE;
    System.out.println( this + " termino llamada." );
    other = null;
}

protected Boolean ring( TwoWay sender ) {
    if ( state != IDLE ) {
        return false;
    }
    state = RINGING;
    System.out.println( this + "RRRRIIIIIIINNNNGGG!!!!" );
    other = sender;
    return true;
}
```

```
}

public void answerCall( String m ) {
    if ( state != RINGING ) {
        return;
    }
    System.out.println( this + " Contesto." );
    state = BUSY;
    SendMessage( m );
}

public String number() {
    return number;
}

public Boolean sendMessage( String m ) {
    if ( state != BUSY ) {
        return false;
    }
    if ( other == null ) {
        return false;
    }
    System.out.println( this + " envia: " + m );
    return other.receiveMessage( m );
}

protected Boolean receiveMessage( String m ) {
    if ( state != BUSY ) {
        return false;
    }
    System.out.println( this + " recibio: " + m );
    return true;
}

public String toString() {
    String [] s = { "IDLE", "BUSY", "RINGING" };

    return super.toString()
        + " #: " + number
        + " s: " + s[state];
}
}
```

Clase HomePhone

```
public class HomePhone extends TwoWay implements Fixed {
    private String address;

    public HomePhone( String serialNum, String num, Exchange e ) {
```

```
super( serialNum, num, e );
}

public String toString() {
    return super.toString() + " A:" + address;
}

public void setAddress( String addr ) {
    address = addr;
}
}
```

Clase Celular

```
public class Celular extends TwoWay implements Mobile {
    private Boolean on;

    public Celular( String serialNum, String number, Exchange e ) {
        super( serialNum, number, e );
        on = false;
    }

    public Boolean dial( String n ) {
        if ( !on ) {
            return false;
        }
        return super.dial( n );
    }

    public Boolean receiveMessage( String m ) {
        if ( !on ) {
            return false;
        }
        return super.receiveMessage( m );
    }

    public void ring() {
        if ( !on ) {
            return;
        }
        super.ring( this );
    }

    public Boolean onOff() {
        on = !on;
        if ( !on ) {
            endCall();
        }
        return on;
    }
}
```

```
}

public Boolean isOn() {
    return on;
}

public String toString() {
    return super.toString() + " on:" + on;
}
}
```

Fuentes

Booch G., *Análisis y diseño orientado a objetos con aplicaciones*, 2^a ed., México, Addison Wesley Longman, 1998.

Ceballos, F. J., Java 2. *Curso de programación*, 4a ed., México, Ra-ma, 2010.

Deitel, P.J. y H.M. Deitel, *Java. Cómo programar*, 7a ed., México, Pearson Prentice Hall, 2008.

Denti, E., *Lucidi del corso Fondamenti di Informatica*, Boloña, LB, Facoltà di Ingegneria, Università degli Studi di Bologna, 2004.

Fowler, M., *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, Nueva York, Addison-Wesley, 2004.

Gómez, M.C., J. Cervantes y A. García, “Metodología para la enseñanza de la programación estructurada”, Orlando, Florida, IX Simposio Iberoamericano en Educación Cibernética e Informática (SIECI), 17-20 de julio, 2012, pp. 218-223.

Joyanes, L. *Programación en Java 2: algoritmos, estructuras de datos y programación orientada a objetos*, México, McGraw Hill, 2002.

Ricci, A., *Lucidi del corso Fondamenti di Informatica*, Cesena, LB, II Facoltà di Ingegneria, Università degli Studi di Bologna, 2005.

The Java Tutorials, Oracle, 2014.

Wirth, Niklaus, “Program Development by Stepwise Refinement”, *Communications of the ACM*, vol. 14, núm. 4 (1971), pp. 221-227.

Glosario

Atributos. Modelan el estado de un objeto, son los datos que contiene el objeto.

Agregación. Es una relación entre clases, en la cual si un objeto de la clase B forma parte de una clase A, se dice que el objeto de la clase B está agregado al objeto de la clase A. Cuando desaparece el objeto de clase A, el de clase B sigue existiendo.

Clase. Es un tipo de dato definido por el programador, específicamente para crear objetos.

Clase abstracta. Sirve para organizar mejor la estructura de un programa. No se pueden instanciar objetos de una clase abstracta, sin embargo, sí se pueden instanciar objetos de las subclases que heredan de una clase abstracta.

Comportamiento. Es la manera en la que el objeto responde a estímulos del exterior.

Composición. Es una relación de entre dos clases A y B, en la que A está compuesta por uno o más objetos de clase B, los objetos de clase B son dependientes de la clase A. Así, cuando desaparece el objeto de clase A, desaparecen también los objetos de clase B.

Estado. Son los datos asociados a un objeto, indican la situación interna del objeto.

Herencia. Es un mecanismo mediante el cual la clase hija o subclase adquiere los atributos y métodos de la clase padre.

Instanciar un objeto. Crear un objeto con los métodos y atributos definidos en una clase.

Método. Es una función que está dentro de una clase. Los métodos definen el comportamiento de un objeto.

Objeto. Es una entidad virtual (o entidad de software) que tiene datos y funciones que simulan las propiedades de un objeto o concepto de la vida real.

POO. Programación orientada a objetos.

Polimorfismo. Es la propiedad que tiene un objeto de clase A, de mostrarse como de clase B o de clase C.

Referencia. Es el identificador de un objeto. En Java no existen los apuntadores, el

concepto de referencia es similar al de apuntador, con la diferencia de que el programador no puede modificar ni acceder directamente al contenido de una referencia.

UML. Unified Modeling Language (Lenguaje Unificado de Modelado).

Introducción a la programación orientada a objetos, se terminó de imprimir en la Ciudad de México en Octubre de 2016, en los talleres de la Imprenta 1200+ ubicada en Andorra 29, Colonia del Carmen Zacahuitzco. La producción editorial estuvo a cargo de Servicios Editoriales y Tecnología Educativa Prometheus S.A. de C.V. En su composición se usaron tipos Minion Pro y Avenir. Se tiraron 100 ejemplares sobre papel bond de 90 kilogramos. La corrección de estilo estuvo a cargo de Hugo A. Espinoza Rubio y el diseño editorial y la portada fueron realizadas por Ricardo López Gómez.