PROGRAMACIÓN ORIENTADA A OBJETOS

José Sánchez Juárez

Agosto 2019-Febrero 2020

II

Índice general

Índice de figuras

Índice de cuadros

Introducción a la programación orientada a objetos

> ARTICULO 1.- El Instituto Politécnico Nacional es la institución educativa del Estado creada para consolidar, a través de la educación, la Independencia Económica, Científica, Tecnológica, Cultural y Política para alcanzar el progreso social de la Nación, de acuerdo con los objetivos Históricos de la Revolución Mexicana, contenidos en la Constitución Política de los Estados Unidos Mexicanos.
>
> Ley orgánica.
> Instituto Politécnico Nacional

El paradigma orientado a objetos es la forma más común de analizar, diseñar,y desarrollar sistemas de aplicación, especialmente muy grandes. Para entender este paradigma, podríamos preguntarnos: ¿que exactamente significa la frase .ºrientado a objetos¿ Buscando en la literatura, etiquetar algo como orientado a objetos implica que el objeto juega un rol central, y elaboramos esta característica como una perspectiva que ve el elemento de una situación dada por descomposición de objetos y relación de objetos. En un sentido amplio, esta idea podría aplicarse a cualquier entorno y los ejemplos de su aplicación pueden aplicarse en negocios, química, ingeniería e incluso filosofía. Nuestro negocio es con la creación de software y, por lo tanto, este libro se concentra en el análisis orientado a objetos, diseño e implementación de sistemas de software. Nuestras situaciones son, por tanto, problemas que son susceptibles a las soluciones de software y los sistemas de software que se crean en respuesta a estos problemas.

El diseño es una actividad complicada en cualquier contexto simple porque hay competencia de intereses, donde se tienen que hacer una selección de criterios en cada paso con información incompleta. Como un resultado, las decisiones son frecuentemente hechas de una combinación de reglas derivadas de experiencias pasadas. El diseño de software no es una excepción de esto, y en el proceso de diseñar un sistema, hay muchos puntos donde tales decisiones tienen que ser hechas. Haciendo selecciones informadas en cualquier campo de actividad se requiere un entendimiento de la filosofía y de las fuerzas que se han formado. Es

por ello apropiado comenzar nuestro estudio de análisis de software orientado a objetos y diseño aplicando la filosofía y el desarrollo en este campo hasta la actualidad. A través del caso de estudio usado en este texto, el lector encontrará ejemplos de como esta guía filosófica nos auxilia para hacer selecciones en todos los pasos.

Este capítulo, por lo tanto, pretende dar al lector una amplia introducción al complejo tema de desarrollo de software orientado a objetos. Comenzamos con una descripción general de la circunstancia que motivaron su desarrollo y por qué se convirtió en el enfoque deseado para desarrollo de software. En el curso de esta discusión, presentamos los conceptos centrales que caracterizan la metodología, cómo este desarrollo ha influido en nuestra visión del software, y algunos de sus pros y contras. Concluimos presentando una breve historia de la evolución del enfoque orientado a objetos.

¿Que es el desarrollo orientado a objetos?

La visión tradicional de un programa de computadora es la de un proceso que ha sido codificado en un formulario que se puede ejecutar en una computadora. Esta visión se originó del hecho de que la primera Las computadoras se desarrollaron principalmente para automatizar un proceso bien definido (es decir, un algoritmo) para el cálculo numérico, y se remonta a las primeras computadoras del programa almacenado. Acuerdo ingly, el proceso de creación de software fue visto como una traducción de una descripción en algunos Lenguaje "natural"para una secuencia de operaciones que podrían ejecutarse en una computadora. Como Muchos argumentarían que este paradigma sigue siendo la mejor manera de introducir la noción de programa. ming a un principiante, pero a medida que los sistemas se vuelven más complejos, su eficacia en el desarrollo Las soluciones se volvieron sospechosas. Este cambio de perspectiva por parte de los desarrolladores de software sucedió durante un período de tiempo y fue alimentado por varios factores, incluido el alto costo de desarrollo y los constantes esfuerzos para encontrar usos para el software en nuevos dominios. Uno podria argumenta con seguridad que las aplicaciones de software desarrolladas en años posteriores tuvieron dos diferencias características:

1. Comportamiento que fue difícil de caracterizar como un proceso.
2. Requisitos de confiabilidad, rendimiento y costo que los desarrolladores originales no enfrentaron.

El enfoque çentrado en el proceso"para el desarrollo de software utilizaba lo que se denomina funciones de arriba hacia abajo descomposición nacional. El primer paso en tal diseño fue reconocer lo que el proceso tenía para entregar (en términos de entrada y salida del programa), que fue seguido por descomposición del proceso en módulos funcionales. Se definieron estructuras para almacenar datos y el cómputo se realizó invocando los módulos, que realizaron algunos cálculos tanto en los elementos de datos almacenados. La vida de un diseño centrado en el proceso fue corta porque cambios en la especificación del proceso (algo relativamente poco común con algo numérico los ritmos en comparación con las aplicaciones comerciales) requirieron un cambio en todo el programa.

Esto a su vez resultó en una incapacidad para reutilizar el código existente sin una sobrecarga considerable. Como resultado, los diseñadores de software comenzaron a analizar sus propios enfoques y también a estudiar firmar procesos y principios que estaban siendo empleados por ingenieros en otras disciplinas. La polinización cruzada de ideas de otras disciplinas de ingeniería comenzó poco después, y surgieron disciplinas de "diseño de software.eingeniería de software". A este respecto, es interesante observar el proceso utilizado para diseñar elementos eléctricos simples. sistemas electromecánicos. Durante varias décadas, ha sido bastante fácil para las personas con conocimiento limitado de los principios de ingeniería diseñar y armar sistemas simples en sus patios y garajes. Tanto es así, que se ha convertido en un pasatiempo que incluso un niño de diez años podría perseguir. Las razones de este éxito son fáciles de ver: diseños fácilmente comprensibles, soluciones similares (estándar) para una serie de problemas, de fácil acceso y bien definidos "Biblioteca"de "bloques de construcción", intercambiabilidad de componentes entre sistemas, etc. Algunos de los pioneros en el campo del diseño de software comenzaron a preguntarse si no podían también diseñe software que utilice dichos componentes "listos para usar". El paradigma orientado a objetos, se podría argumentar que realmente ha evolucionado en respuesta a esta perspectiva. Hay, por supuesto, varias diferencias con el proceso de diseño del hardware (inevitable, porque la naturaleza del software es fundamentalmente diferente del hardware), pero se pueden establecer paralelos entre muchas de las definiciones de las características del diseño de hardware y lo que hoy defiende el buen software: Diseño recomendado. Esta metodología, como veremos en los capítulos siguientes, nos proporciona con un proceso paso a paso para el diseño de software, un lenguaje para especificar el resultado de cada paso del proceso para que podamos pasar sin problemas de una etapa a la siguiente, la capacidad para reutilizar diseños anteriores, soluciones estándar que se adhieren a principios de diseño bien razonados e, incluso, la capacidad de corregir gradualmente un diseño deficiente sin romper el sistema. La filosofía general aquí es definir un sistema de software como una colección de objetos de varios tipos que interactúan entre sí a través de interfaces bien definidas. A diferencia de un duro componente de software, un objeto de software puede diseñarse para manejar múltiples funciones y puede por lo tanto, participe en varios procesos. Un componente de software también es capaz de almacenar datos, que agrega otra dimensión de complejidad al proceso. La manera en que todos de esto se ha alejado de la visión tradicional orientada al proceso es que en lugar de implementar al realizar un proceso completo de principio a fin y definir las estructuras de datos necesarias en el camino, primero analice todo el conjunto de procesos y, a partir de este, identifique el software necesario con ponentes de cada componente que representa una abstracción de datos y está diseñado para almacenar información junto con procedimientos para manipular lo mismo. La ejecución de los procesos originales es luego desglosado en varios pasos, cada uno de los cuales puede asignarse lógicamente a uno de los componentes de software, los componentes también pueden comunicarse entre sí según sea necesario para completar el proceso.

Conceptos clave para el diseño orientado a objetos

Durante el desarrollo de este paradigma, como cabría esperar, varias ideas y los ensayos fueron juzgados y descartados. Con los años, el campo se ha estabilizado para que tengamos presente con seguridad las ideas clave cuya solidez ha resistido la prueba del tiempo.

0.0.1. El rol central de los objetos

La orientación a objetos, como su nombre lo indica, convierte a los objetos en la pieza central del diseño de software. El diseño de los sistemas anteriores se centró en los procesos, que eran susceptibles de cambio, y cuando se produjo este cambio, muy poco del viejo sistema era reutilizable". La noción de un objeto se centra en una pieza de datos y las operaciones (o métodos) podrían usarse para modificarlo. Esto hace posible la creación de una abstracción que es muy estable ya que no depende de los requisitos cambiantes de la aplicación. La ejecución de cada proceso depende en gran medida de los objetos para almacenar los datos y proporcionar operaciones necesarias; con algo de trabajo adicional, todo el sistema se .[en]sambla.[a] partir de objetos.

0.0.2. La noción de clase

Las clases permiten que un diseñador de software vea los objetos como diferentes tipos de entidades. Visitar los objetos de esta manera nos permiten utilizar los mecanismos de clasificación para clasificar estos tipos, definir jerarquías y comprometerse con las ideas de especialización y generalización de objetos.

0.0.3. Especificación abstracta de funcionalidad

En el curso del proceso de diseño, el ingeniero de software especifica las propiedades de los objetos. (y, por implicación, las clases) que necesita un sistema. Esta especificación es abstracta ya que no impone restricciones sobre cómo se logra la funcionalidad. Esta especificación, llamada interfaz o clase abstracta, es como un contrato para el implementador lo que también facilita la verificación formal de todo el sistema.

0.0.4. Un lenguaje para definir el sistema

El lenguaje de modelado unificado (UML) ha sido elegido por consenso como herramienta estándar para describir los productos finales de las actividades de diseño. Los documentos generados en este lenguaje se puede entender universalmente y, por lo tanto, es análogo a los "planosütilizados en otras disciplinas de ingeniería.

0.0.5. Soluciones estándar

La existencia de una estructura de objetos facilita la documentación de soluciones estándar, llamadas patrones de diseño. Las soluciones estándares se encuentran en todas las etapas del desarrollo de software, pero los patrones de diseño son quizás la forma más común de reutilización de soluciones.

0.0.6. Un proceso de análisis para modelar un sistema

La orientación a objetos nos proporciona una forma sistemática de traducir una especificación funcional a un diseño conceptual. Este diseño describe el sistema en términos de clases conceptuales de que los pasos posteriores del proceso de desarrollo generan las clases de implementación que constituyen el software terminado.

0.0.7. La notación de extandabilidad y adaptabilidad

El software tiene una flexibilidad que normalmente no se encuentra en el hardware, y esto nos permite modificar las entidades existentes en pequeñas formas de crear otras nuevas. la herencia, que crea un nuevo clase descendiente que modifica las características de una clase existente (ancestro) y su composición, que utiliza objetos que pertenecen a clases existentes como elementos para constituir una nueva clase, son mecanismos que permiten tales modificaciones con clases y objetos.

Otros conceptos relacionados

A medida que se desarrolló la metodología orientada a objetos, la ciencia del diseño de software progresó también, y se identificaron varias propiedades de software deseables. No es lo suficientemente central como para ser llamados conceptos orientados a objetos, estas ideas están estrechamente vinculadas a ellos y son quizás mejor entendidas debido a estos desarrollos.

0.0.8. Diseño modular y encapsulamiento

La modularidad se refiere a la idea de armar un sistema grande desarrollando un número de componentes distintos de forma independiente y luego integrarlos para proporcionar la requerida funcionalidad, este enfoque, cuando se usa correctamente, generalmente hace que los módulos individuales relativamente simples y, por lo tanto, el sistema sea más fácil de entender que uno diseñado como estructura monolítica. En otras palabras, dicho diseño debe ser modular. La función del sistema debe ser proporcionada por una serie de módulos bien diseñados y cooperantes. Cada modulo obviamente, debe proporcionar cierta funcionalidad claramente especificada por una interfaz. La interfaz también define cómo otros componentes pueden interactuar o comunicarse con el módulo. Nos gustaría que un módulo especifique claramente lo que hace, pero no exponga su implementación. Esta separación de preocupaciones da lugar a la noción de encapsulación, lo que significa que el módulo oculta detalles de su implementación de agentes externos. Los tipos de datos abstractos (ADT),

la generalización de tipos de datos primitivos como enteros y caracteres, es un ejemplo de aplicación de encapsulación. El programador especifica la colección de operaciones en el tipo de datos y las estructuras de datos que se necesitan para el almacenamiento de datos. Los usuarios del ADT realizan las operaciones sin preocuparse por la implementación.

0.0.9. Cohesión y acoplamiento

Cada módulo proporciona cierta funcionalidad; la cohesión de un módulo nos dice qué tan bien están las entidades dentro de un módulo para trabajar juntas para proporcionar esta funcionalidad. La cohesión es una medida de cuán enfocadas están las responsabilidades de un módulo. Si las responsabilidades de un módulo son no relacionadas o variadas para utilizar diferentes conjuntos de datos, se reduce la cohesión. Los módulos altamente cohesivos tienden a ser más confiables, reutilizables y comprensibles que los menos cohesivos. Al aumentar la cohesión, nos gustaría que todos los componentes contribuyan a algunas bien definidas responsabilidades del módulo. Esta puede ser una tarea bastante desafiante. En contraste, el peor enfoque sería asignar arbitrariamente entidades a los módulos, lo que resulta en un módulo cuyos constituyentes no tienen una relación obvia.

El acoplamiento se refiere a la dependencia de los módulos entre sí. El hecho mismo de que nosotros al dividir un programa en múltiples módulos se introduce algún acoplamiento en el sistema. Podría producirse pling debido a varios factores: un módulo puede referirse a variables definidas en otro módulo o un módulo puede llamar a métodos de otro módulo y usar el valor de retorno. La cantidad de acoplamiento entre módulos puede variar. En general, si los módulos no dependen de la implementación de los demás, es decir, los módulos dependen solo de la información publicada en las superficies de otros módulos y no en sus componentes internos, decimos que el acoplamiento es bajo. En En tales casos, los cambios en un módulo no requerirán cambios en otros módulos siempre y cuando ya que las interfaces en sí no cambian. El acoplamiento bajo nos permite modificar un modulo sin preocuparse por las ramificaciones de los cambios en el resto del sistema. Por el contrario, un alto acoplamiento significa que los cambios en un módulo requerirían cambios en otros módulos, que pueden tener un efecto dominó y también dificultar su comprensión del código.

0.0.10. Modificabilidad y comprobabilidad

Un componente de software, a diferencia de su contraparte de hardware, puede modificarse fácilmente en pequeños formas. Esta modificación se puede hacer para cambiar tanto la funcionalidad como el diseño. La habilidad de cambiar la funcionalidad de un componente permite que los sistemas sean más adaptables; los avances en la orientación a objetos han establecido estándares más altos para la adaptabilidad. Mejorando el el diseño a través del cambio incremental se logra refactorizando, nuevamente un concepto que debe su origen al desarrollo del enfoque orientado a objetos. Hay algún riesgo asociado con actividades de ambos tipos; y en ambos casos, la organización del sistema en términos de objetos y clases ha ayudado a desarrollar procedimientos sistemáticos que mitigan el riesgo.

La capacidad de prueba de un concepto, en general, se refiere tanto a la falsabilidad como a la facilidad con la que pueden encontrar contra ejemplos y la factibilidad práctica de reproducir dicho contra ejemplo. En el contexto de los sistemas de software, simplemente se puede expresar como la facilidad con la que se puede encontrar errores en un software y en qué medida la estructura del sistema facilita la detección de errores. Varios conceptos en las pruebas de software (por ejemplo, la idea de las pruebas unitarias) deben su importancia a los conceptos que surgieron del desarrollo del paradigma orientado a objetos.

Beneficios y desventajas del paradigma

Desde un punto de vista práctico, es útil examinar cómo la metodología orientada a objetos tiene modificado el panorama del desarrollo de software. Como con cualquier desarrollo, tenemos pros y contras. Las ventajas que se enumeran a continuación son en gran medida consecuencia de las ideas presentadas en las secciones anteriores.

1. Los objetos a menudo reflejan entidades en los sistemas de aplicación. Esto hace que sea más fácil para que un firmante para llegar a clases en el diseño. En un diseño orientado a procesos, es mucho más difícil encontrar una conexión que pueda simplificar el diseño inicial.

2. La orientación a objetos ayuda a aumentar la productividad mediante la reutilización del software existente. La herencia hace que sea relativamente fácil ampliar y modificar la funcionalidad proporcionada por un clase. Los diseñadores de idiomas a menudo suministran bibliotecas extensas que los usuarios pueden ampliar.

3. Es más fácil acomodar los cambios. Una de las dificultades con el desarrollo de aplicaciones está cambiando los requisitos. Con cierto cuidado durante el diseño, es posible para aislar las diferentes partes de un sistema en clases.

4. La capacidad de aislar cambios, encapsular datos y emplear modularidad reduce los riesgos involucrados en el desarrollo del sistema.

Las ventajas anteriores no vienen sin una etiqueta de precio. Quizás la víctima número uno del paradigma es la eficiencia. El proceso de desarrollo orientado a objetos introduce muchos capas de software, y esto ciertamente aumenta los gastos generales. Además, la creación de objetos y la destrucción es cara. Las aplicaciones modernas tienden a presentar una gran cantidad de objetos que interactúan entre sí de manera compleja y al mismo tiempo admiten un usuario con interfaz visual. Esto es cierto si se trata de una aplicación bancaria con numerosos objetos de cuenta o un videojuego que a menudo tiene una gran cantidad de objetos. Los objetos tienden a tener complejos asociaciones, que pueden dar como resultado una no localidad, lo que lleva a tiempos de acceso de memoria deficientes. Programadores y diseñadores educados en otros paradigmas, generalmente en el paradigma imperativo, les resulta difícil aprender y utilizar principios orientados a objetos. En llegar a clases, los diseñadores inexpertos pueden confiar demasiado en las entidades de la aplicación del sistema,

terminando con sistemas que no son adecuados para su reutilización. Los programadores también necesitan acclimatización de algunas personas que estiman que un programador tarda hasta un año en comenzar sentirse cómodo con estos conceptos. Algunos investigadores opinan que los entornos de gramática tampoco se han mantenido al día con la investigación en capacidades lingüísticas. Ellos consideran que muchos de los editores y las instalaciones de prueba y depuración siguen siendo fundamentalmente orientado al paradigma imperativo y no respalda directamente muchos de los avances tales como patrones de diseño.

Historia del paradigma

La historia del enfoque de programación orientada a objetos se remonta a la idea de ADT y el concepto de objetos en el lenguaje de programación Simula 67, desarrollado en la década de 1960 para realizar simulaciones. El primer verdadero lenguaje de programación orientado a objetos. que apareció antes de que la comunidad de desarrollo de software más grande fuera Smalltalk en 1980, desarrollado en Xerox PARC. Smalltalk utilizaba objetos y mensajes como base para la comunicación. Las clases pueden ser creadas y modificadas dinámicamente. La mayor parte del vocabulario en el paradigma orientado a objetos se originó a partir de este lenguaje. A fines de la década de 1970, Bjarne Stroustrup, que realizaba un doctorado en Inglaterra. Necesitaba un lenguaje para hacer simulación de sistemas distribuidos. Desarrolló un lenguaje indicador basado en el concepto de clase en Simula, pero este lenguaje no fue particularmente eficaz. Sin embargo, él persiguió su intento y desarrolló un lenguaje orientado a objetos en los laboratorios Bell como un derivado de C, que se convertiría en uno de los más exitosos lenguajes de programación, C ++. El lenguaje fue estandarizado en 1997 por el Instituto Nacional de Normas (ANSI) estadounidense. La década de 1980 vio el desarrollo de varios otros idiomas como ObjectLisp, CommonLisp, Common Lisp Object System (CLOS) y Eiffel. La creciente popularidad de la modelación orientado a objetos también impulsó cambios en el lenguaje Ada, originalmente patrocinado por el Departamento de Defensa de los Estados Unidos en 1983. Esto resultó en Ada 9x, una extensión de Ada 83, con conceptos orientados a objetos que incluyen herencia, polimorfismo y enlace dinámico. La década de 1990 vio dos grandes eventos. Uno fue el desarrollo de la programación Java, lenguaje de 1996. Java parecía ser un derivado de C ++, pero muchos de los controvertidos y los conceptos problemáticos en C ++ se eliminaron en él. Aunque era relativamente un lenguaje simple cuando se propuso originalmente, Java ha sufrido adiciones sustanciales, en el futuro las versiones que lo convertiran en un lenguaje moderadamente difícil. Java también viene con un impresionante colección de bibliotecas (llamadas paquetes) para soportar el desarrollo de aplicaciones. Un segundo evento principalmente fue la publicación del libro Design Patterns de Gamma et al. en 1994. El libro consideró preguntas de diseño específicas (23 de ellas) y proporcionó enfoques generales para resolver usando construcciones orientadas a objetos. El libro (como también el enfoque que defendió) fue un gran éxito ya que tanto los profesionales como los académicos pronto reconocieron su importancia. Los últimos años vieron la aceptación de algunos lenguajes dinámicos orientados a objetos que fueron desarrollados en la década de 1990. Los lenguajes dinámicos permiten a los usuarios una mayor

flexibilidad, por ejemplo la capacidad de agregar dinámicamente un método a un objeto en tiempo de ejecución. Uno de esos idiomas es Python, que se puede utilizar para resolver una variedad de aplicaciones, incluido el programa web ming, bases de datos, cómputos científicos y numéricos y redes. Otra lenguaje dinámico es el lenguaje, Ruby, que está aún más orientado a objetos, ya que todo en el lenguaje, incluso los números y los tipos primitivos, son un objeto.

Discusión y lectura adicional

En este capítulo, hemos dado una introducción al paradigma orientado a objetos. Los conceptos centrales orientados a objetos como clases, objetos e interfaces serán abordados en los próximos tres capítulos. La cohesión y el acoplamiento, que son los principales problemas de diseño de software, ser temas recurrentes para la mayor parte del texto. Se aconseja al lector que aprenda o se actualice los conceptos no orientados a objetos. Del lenguaje Java lea el Apéndice A antes de pasar al siguiente capítulo. Vale la pena y es divertido leer una breve historial de los lenguajes de programación desde un texto estándar sobre el tema como Sebesta [33]. El lector también puede encontrar útil obtener las perspectivas de los diseñadores de los lenguajes orientados a objetos (como el dado en C ++ por Stroustrup [37]).

0.0.11. Ejercicios

1. Identifique a los jugadores que tendrían interés en el proceso de desarrollo de software. ¿Cuales son las preocupaciones de cada uno? ¿Cómo se beneficiarían del modelo orientado a objetos?

2. Piense en algunas empresas comunes y en las actividades en las que participan los desarrolladores de software. ¿Cuáles son los conjuntos de procesos que les gustaría automatizar? ¿Hay alguno que necesite software solo para un proceso?

3. ¿Cómo apoya el modelo orientado a objetos la noción de ADT y encapsulación?

4. Considere una aplicación con la que esté familiarizado, como un sistema universitario. Dividir las entidades de esta aplicación en grupos, identificando así las clases.

5. En la pregunta 4, supongamos que ponemos todo el código (correspondiente a todas las clases) en una sola clase. ¿Qué pasa con la cohesión y el acoplamiento?

6. ¿Cuáles son los beneficios de aprender patrones de diseño?

[2]

Bases de la programación orienta a objetos

En el último capítulo, vimos que la estructura del programa fundamental en un programa orientado a objetos es el objeto. También describimos el concepto de una clase, que es similar a los ADT ya que puede usarse para crear objetos de tipos que el lenguaje no admite directamente. En el último capítulo, vimos que la estructura fundamental del programa en un objeto orientado. En este capítulo, describimos en detalle cómo construir una clase. Utilizaremos la gramática del lenguaje Java (como haremos a lo largo del libro). Vamos a presentar el Unified Modeling Language (UML), que es una notación para describir el diseño de sistemas orientados a objetos. También discutimos las interfaces, un concepto que nos ayuda a especificar los requisitos del programa y demostrar sus usos.

0.0.12. Las bases

Para entender la noción de objetos y clases, comenzamos con una analogía. Con respecto a un coche el fabricante decide construir un auto nuevo, se dedica un esfuerzo considerable en una variedad de actividades antes de que el primer automóvil salga de las líneas de montaje. Éstos incluyen:

1. Identificación de la comunidad de usuarios para el automóvil y evaluación de las necesidades del usuario. Para esto, el fabricante puede formar un equipo.

2. Después de evaluar los requisitos, el equipo puede expandirse para incluir ingenieros automotrices y otros especialistas que elaboran un diseño preliminar.

3. Se puede utilizar una variedad de métodos para evaluar y refinar el diseño inicial (el equipo puede tener experiencia en la construcción de un vehículo similar): se pueden construir prototipos, simulaciones y se puede realizar un análisis matemático.

Quizás después de meses de tal actividad, se complete el proceso de diseño. Otro paso que es necesario realizar es la construcción de la planta donde se producirá el automóvil. Se debe establecer una línea de montaje y contratar personas. Después de tales pasos, la compañía está lista para producir los automóviles. El diseño ahora se reutiliza para muchos tiempos de fabricación. Por supuesto, el diseño puede tener que ser ajustado durante el proceso según las observaciones de la empresa y los comentarios de los usuarios. El desarrollo de

sistemas de software a menudo siguen un patrón similar. Las necesidades del usuario tienen que ser evaluadas, se debe hacer un diseño y luego se debe construir el producto. Desde el punto de vista de los sistemas orientados a objetos, un aspecto diferente de la fabricación de automóviles El proceso de turing es importante. El diseño de un determinado tipo de automóvil requerirá tipos específicos del motor, la transmisión, el sistema de frenos, etc., y cada una de estas partes en sí misma tiene su diseño propio (diseño original), plantas de producción, etc. En otras palabras, la compañía sigue la misma filosofía en la fabricación de las piezas individuales que en la producción de el coche. Por supuesto, algunas partes se pueden comprar a los fabricantes, pero a su vez siguen el mismo enfoque. Dado que la actividad de diseño es costosa, un fabricante reutiliza el diseño para fabricar las partes o los autos. El enfoque anterior se puede comparar con el diseño de sistemas orientados a objetos que están compuestos de muchos objetos que interactúan entre sí. A menudo, estos objetos representan los jugadores de la vida real y sus interacciones representan interacciones de la vida real. Como el diseño de un auto es una colección de los diseños individuales de sus partes y un diseño de la interacción de estas partes, el diseño de un sistema orientado a objetos consiste en diseños de sus partes constituyentes y sus interacciones. Por ejemplo, un sistema bancario podría tener un conjunto de objetos que representan a los clientes, otro conjunto de objetos que representan cuentas y un tercer conjunto de objetos que corresponden a préstamos. Cuando un cliente realmente hace un depósito en su cuenta en la vida real, el sistema actúa sobre el objeto de la cuenta correspondiente para imitar el depósito en el software. Cuando un cliente solicita un préstamo, se crea un nuevo objeto de préstamo y se conecta al objeto del cliente; cuando un pago se realiza sobre el préstamo, el sistema actúa sobre el objeto del préstamo correspondiente. Obviamente, estos objetos tienen que ser creados de alguna manera. Cuando un nuevo cliente ingresa al sistema, deberíamos poder crear un nuevo objeto de cliente en software. Esta entidad de software, el objeto del cliente debe tener todas las características relevantes del cliente de la vida real. Por ejemplo, debería ser posible asociar el nombre y la dirección del cliente con este objeto sin embargo, los atributos del cliente que no son relevantes para el banco no serán representado en software. Como ejemplo, es difícil imaginar que un banco esté interesado en si un cliente es diestro; por lo tanto, el sistema de software no tendrá este atributo.

DEFINICIÓN 1 (Atributo.) *Un atributo es una propiedad que asociamos con un objeto; sirve para describir el objeto que contiene algún valor que se requiere para el procesamiento. [9] .*

El mecanismo de clase en lenguajes orientados a objetos proporcionan una forma de crear tales objetos. Una clase es un diseño que se puede reutilizar cualquier cantidad de veces para crear objetos. Por ejemplo, considerar un sistema orientado a objetos para una universidad. Hay objetos de estudiante, objetos de instructor, objetos de miembros del personal, etc. Antes de crear tales objetos, creamos las clases que sirven como planos para estudiantes, instructores, miembros del personal y cursos de la siguiente manera:

Listing 1: Clases.

```
1  public class Student {
2  // codigo pera implementar un solo estudiante
3  }
4  public class Instructor {
5  // codigo pera implementar un solo instructor
6  }
7  public class StaffMember {
8  // codigo pera implementar un solo miembro del staff
9  }
10 public class Course {
11 // codigo pera implementar un solo curso
12 }
```

Las definiciones anteriores muestran cómo crear cuatro clases, sin dar ningún detalle. (Nosotros deberíamos poner los detalles donde hemos dado comentarios.) La clase de token es una palabra clave que dice que estamos creando una clase y que el siguiente token es el nombre de la clase. De este modo, hemos creado cuatro clases: Estudiante, Instructor, StaffMember y Curso. El corchete izquierdo () significa el comienzo de la definición de la clase y el corchete derecho correspondiente () finaliza la definición. El token público es otra palabra clave que hace que la clase correspondiente esté disponible en todo el sistema de archivos. Antes de ver cómo poner los detalles de la clase, veamos cómo crear objetos usando estas clases. El proceso de crear un objeto también se llama instanciación. Cada clase introduce un nuevo nombre de tipo. Así estudiante, instructor, miembro del personal y Curso son los tipos que hemos introducido. El siguiente código crea una instancia de un nuevo objeto de tipo Estudiante.

Listing 2: Instancia de un nuevo objeto de tipo Estudiante.

```
1  new Student();
```

El nuevo operador hace que el sistema asigne un objeto de tipo Estudiante con suficiente almacenamiento para guardar información sobre un alumno. El operador devuelve la dirección de la ubicación que contiene este objeto. Esta dirección se denomina referencia. La declaración anterior se puede ejecutar cuando tenemos un nuevo estudiante admitido en la Universidad. Una vez que instanciamos un nuevo objeto, debemos almacenar su referencia en algún lugar, por lo que que podemos usar más tarde de alguna manera apropiada. Para esto, creamos una variable de tipo Estudiante.

Listing 3: Crear la variable estudiante.

```
1  Student harry;
```

Tenga en cuenta que la definición anterior simplemente dice que Harry es una variable que puede almacenar referencias a objetos de tipo Estudiante. Por lo tanto, podemos escribir

Listing 4: La variable harry.

```
1  harry = new Student();
```

No podemos escribir

Listing 5: No se use.

```
1 harry = new Instructor();
```

porque Harry es del tipo Estudiante, que no tiene ninguna relación (en la medida en que la clase decida a la de Instructor, que es del tipo de objeto creado del lado derecho de la tarea. Cada vez que instanciamos un nuevo objeto, debemos recordar la referencia a ese objeto en algún lado. Sin embargo, no es necesario que por cada objeto que instanciamos, declaremos una variable diferente para almacenar su referencia. Si ese fuera el caso, la programación sería tediosa. Vamos a ilustrar dando una analogía. Cuando un estudiante conduce a la escuela para tomar una clase, ella se ocupa solo de un número relativamente pequeño de objetos: los controles del automóvil, la carretera, los vehículos cercanos (y a veces sus ocupantes, aunque no siempre educadamente) y el tránsito, señales y signos. (Algunos también pueden tratar con un teléfono celular, ¡lo cual no es una buena idea!) Hay muchos otros objetos que el conductor (estudiante) conoce, pero que no está tratando con ellos. Del mismo modo, mantenemos las referencias a un número relativamente pequeño de objetos en nuestros programas. Cuando surge la necesidad de acceder a otros objetos, utilizamos las referencias que ya tenemos para cubrirlos. Por ejemplo, supongamos que tenemos una referencia a un objeto Student. Ese objeto puede tener un atributo que recuerda al asesor del alumno, un objeto Instructor. Si es necesario averiguar el asesor de un alumno determinado, podemos consultar el correspondiente objeto del alumno para obtener el objeto Instructor. Un solo objeto Instructor puede tener atributos que recuerdan todos los consejos del instructor correspondiente.

Implementando clases

En esta sección damos algunos de los conceptos básicos para crear clases. Centrémonos en la clase estudiante que inicialmente codificamos como

Listing 6: Clase estudiante.

```
1 public class Student {
2 // codigo pera implementar un solo student
3 }
```

Ciertamente nos gustaría la capacidad de dar un nombre a un estudiante: dado un objeto de estudiante, nosotros debemos poder especificar que el nombre del alumno es "Tom.º "Jane.º, en general, alguna cuerda, esto a veces se conoce como un comportamiento del objeto. Podemos pensar en los objetos de los estudiantes que tienen el comportamiento que responden al asignar un nombre. Para este propósito, modificamos el código de la siguiente manera.

Listing 7: Dar nombre a un objeto.

```
1 public class Student {
2 // codigo para hacer otras cosas
3 public void setName(String studentName) {
4 // codigo para recordar el nombre
5 }
```

```
6 }
```

El código que agregamos se llama método. El nombre del método es setName. Un método es como un procedimiento o función en programación imperativa en que es una unidad de código que no se activa hasta que se invoca. Nuevamente, como en el caso de procedimientos y funciones, los métodos aceptan parámetros (separados por comas en Java). Cada parámetro establece el tipo de parámetro esperado. Un método puede no devolver nada (como es el caso aquí) o devolver un objeto o un valor de un tipo primitivo. Aquí hemos puesto vacío delante del significado del nombre del método que el método no devuelve nada. Los corchetes izquierdo y derecho comienzan y finalizan el código eso define el método. A diferencia de las funciones y procedimientos, los métodos generalmente se invocan a través de objetos. El método setName se define dentro de la clase Student e invoca un objetos de tipo Estudiante.

Listing 8: Crear otra clase.

```
1 Student aStudent = new Student();
2 aStudent.setName("Ron");
```

El método setName () se invoca en ese objeto al que hace referencia un estudiante. Intuitivamente El código dentro de ese método debe almacenar el nombre en alguna parte. Recuerda que cada objeto Se le asigna su propio almacenamiento. Esta pieza de almacenamiento debe incluir espacio para recordar el nombre del alumno Embellecemos el código de la siguiente manera.

Listing 9: Crear otra clase.

```
1 public class Student {
2 private String name;
3 public void setName(String studentName) {
4 name = studentName;
5 }
6 public String getName() {
7 return name;
8 }
9 }
```

Dentro de la clase hemos definido el nombre de la variable de tipo String. Se llama un campo.

DEFINICIÓN 2 (Campo.) *Un campo es una variable definida directamente dentro de una clase y corresponde a un atributo Cada instancia del objeto tendrá almacenamiento para el campo.*

Examinemos el código dentro del método setName. Toma en un parámetro, studentName, y asigna el valor en ese objeto String al nombre del campo.

Es importante comprender cómo Java usa el campo de nombre. Cada objeto de tipo El estudiante tiene un campo llamado nombre. Invocamos el método setName () en el objeto referido por un estudiante. Como aStudent tiene el nombre del campo e invocamos el método en un estudiante, la referencia al

nombre dentro del método actuará en el campo de nombre de un estudiante. El método getName () recupera el contenido del campo de nombre y lo devuelve. Para ilustrar esto más a fondo, considere dos objetos de tipo Estudiante.

Listing 10: Crear otra clase.

```
Student student1 = new Student();
Student student2 = new Student();
student1.setName("John");
student2.setName("Mary");
System.out.println(student1.getName());
System.out.println(student2.getName());
```

Se puede acceder a los miembros (campos y métodos por ahora) de una clase escribiendo

Listing 11: Crear otra clase.

```
<object-reference>.<member-name>
```

El objeto al que se refiere student1 tiene su campo de nombre establecido en "John", mientras que el objeto referido por student2 tiene su campo de nombre establecido en "Mary". El nombre del campo en el código

Listing 12: Crear otra clase.

```
name = studentName;
```

se refiere a diferentes objetos en diferentes instancias y, por lo tanto, diferentes instancias de campos. Escribamos un programa completo usando el código anterior.

Listing 13: Crear otra clase.

```
public class Student {
// code
private String name;
public void setName(String studentName) {
name = studentName;
}
public String getName() {
return name;
}
public static void main(String[] s) {
Student student1 = new Student();
Student student2 = new Student();
student1.setName("John");
student2.setName("Mary");
System.out.println(student1.getName());
System.out.println(student2.getName());
}
}
```

La palabra clave public delante del método setName () hace que el método esté disponible donde esté disponible el objeto. Pero, ¿qué pasa con la palabra

clave privada frente a ¿nombre del campo? Significa que solo se puede acceder a esta variable desde el código dentro de la clase Estudiante. Desde la linea

Listing 14: Crear otra clase.

```
1 name = studentName;
```

está dentro de la clase, el compilador lo permite. Sin embargo, si escribimos

Listing 15: Crear otra clase.

```
1 Student someStudent = new Student();
2 someStudent.name = "Mary";
```

fuera de la clase, el compilador generará un error de sintaxis. Como regla general, los campos a menudo se definen con el especificador de acceso privado y el método Los anuncios suelen hacerse públicos. La idea general es que los campos denotan el estado del objeto. y que el estado solo se puede cambiar interactuando a través de métodos predefinidos que denotar el comportamiento del objeto. Por lo general, esto ayuda a preservar la integridad de los datos. Sin embargo, en el ejemplo actual, es difícil argumentar que la consideración de integridad de datos juega un rol en hacer que el nombre sea privado porque todo lo que hace el método setName es cambiar el campo de nombre Sin embargo, si quisiéramos hacer algunas comprobaciones antes de cambiar el estudiante nombre (que no debería suceder tan a menudo), esto nos da una forma de hacerlo. Si hubiéramos guardado nombre público y otros codificados para acceder directamente al campo, haciendo que el campo sea privado más tarde rompería su código. Para un uso más justificado del privado, considere el promedio de calificaciones (GPA) de un estudiante. Claramente, necesitamos hacer un seguimiento del GPA y necesitamos un campo para ello. GPA no es algo eso se cambia arbitrariamente: cambia cuando un estudiante obtiene una calificación para un curso. Entonces haciendo su público podría conducir a problemas de integridad porque el campo puede ser cambiado inadvertidamente por código incorrecto escrito afuera. Por lo tanto, codificamos de la siguiente manera.

Listing 16: Crear otra clase.

```
1  public class Student {
2  // fields to store the classes the student has
3  registered for.
4  private String name;
5  private double gpa;
6  public void setName(String studentName) {
7  name = studentName;
8  }
9  public void addCourse(Course newCourse) {
10 // code to store a ref to newCourse in the Student
11 object.
12 }
13 private void computeGPA() {
14 // code to access the stored courses, compute and
15 set the gpa
16 }
```

```
17 public double getGPA() {
18 return gpa;
19 }
20 public void assignGrade(Course aCourse, char newGrade) {
21 // code to assign newGrade to aCourse
22 computeGPA();
23 }
24 }
```

Ahora escribimos código para utilizar la idea anterior.

Listing 17: Crear otra clase.

```
1 Student aStudent = new Student();
2 Course aCourse = new Course();
3 aStudent.addCourse(aCourse);
4 aStudent.assignGrade(aCourse, B);
5 System.out.println(aStudent.getGPA());
```

El código anterior crea un objeto Estudiante y un objeto Curso. Llama al **addCourse** método en el alumno, para agregar el curso a la colección de cursos tomados por el alumno, y luego llama a **asignarGrade**. Tenga en cuenta los dos parámetros: aCourse y 'B'. Lo implícito lo que significa es que el estudiante ha completado el curso (aCourse) con una calificación de "B". los El código en el método debe calcular el nuevo GPA para el estudiante usando la información presumiblemente en el curso (como el número de créditos) y el número de puntos para una calificación de B'.

Constructores

La clase Estudiante tiene un método para establecer el nombre de un estudiante. Aquí ponemos el nombre del alumno después de crear el objeto. Esto es algo antinatural. Ya que cada estudiante tiene un nombre, cuando creamos un objeto de estudiante, probablemente también sepamos el nombre del estudiante. Sería conveniente almacenar el nombre del alumno en el objeto a medida que lo creamos. objeto. Para ver hacia dónde nos dirigimos, considere las siguientes declaraciones de variables de primitivas tipos de datos.

Listing 18: Crear otra clase.

```
1 int counter = 0;
2 double final PI = 3.14;
```

Ambas declaraciones almacenan valores en las variables a medida que se crean las variables. En el otro mano, el objeto Estudiante, cuando se crea, tiene un cero en cada bit de cada campo. Java y otros lenguajes orientados a objetos permiten la inicialización de campos utilizando qué se llaman constructores.
[1]

Un constructor es un método parecido a una entidad que es llamada cuando un objeto es instanciado. La siguiente instanciación llama a un constructor llamado **Car** que tiene tres parametros un **String**, **int** y **String**.

[Instancia en java.]
new Car(”Porsche”, 2006, ”beige”)

Un ejemplo de constructor es el siguiente:

Listing 19: Ejemplo de constructor en java [3].

```
1  public Car(String m, int y, String c) {
2          this.make = m;
3          this.year = y;
4          this.color = c;
5  }
```

Otro ejemplo de constructor es el siguiente:

Listing 20: Otro ejemplo de constructor en java [3].

```
1  public class Car4 {
2          private String make;
3          private int year;
4          private String color;
5          // car's make
6          // car's manufacturing year
7          // car's primary color
8  //*****************************************
9          public Car4(String m, int y, String c) {
10         this.make = m;
11         this.year = y;
12         this.color = c;
13 } // end constructor
14
15         public String getMake() {
16                 return this.make;
17         } // end getMake
18 } // end class Car4
```

[3]
La definición de constructor se presenta en el siguiente recuadro azul:

DEFINICIÓN 3 (Constructor.) *Un constructor es parecido a un método en el que se puede tener un especificador de acceso (como público o privado), un nombre, parámetros y código ejecutable. Sin embargo, los constructores, tienen las siguientes diferencias o características especiales.*

figura **??** [9] .

Algorithm 0.1: Reconocedor de palabras

```
1  c = SigCar();
2  if c == 'n' then
3  |  c = SigCar();
4  |  if c == 'e' then
5  |  |  c = SigCar();
6  |  |  if c == 'w' then
7  |  |  |  EsEnPan(Palabra Aceptada);
8  |  |  else
9  |  |  |  EsEnPan(Palabra No Aceptada);
10 |  else
11 |  |  EsEnPan(Falla);
12 else
13 |  EsEnPan(Falla);
```

Listing 21: Clase manejadora en java [3].

```
1  /**************************************************
2  * Car4Driver.java
3  * Dean & Dean
4  *
5  * This class is a demonstration driver for the Car4 class.
6  **************************************************/
7  public class Car4Driver
8  {
9          public static void main(String[] args)
10         {
11                 Car4 allexCar = new Car4("Porsche",
12                  2006, "beige");
13                 Car4 latishaCar = new Car4("Saturn",
14                  2002, "red");
15
16                 System.out.println(allexCar.getMake());
17         } // end main
18 } // end class Car4Driver
```

Operador de ámbito

Este operador permite el acceso a las variables globales. Por ejemplo :: count permite el acceso a la versión global de la variable count. El siguiente programa muestra esta característica:

Listing 22: Operador de ámbito en C++.

```
1  #include <iostream>
2
```

```
3  using namespace std;
4  int m=10;
5
6  int main(){
7
8          int m=20;
9
10         {
11
12                 int k = m;
13                 int m = 30;
14
15                 cout << "Estamos dentro del bloque \n";
16                 cout << "k = " << k << "\n" ;
17                 cout << "m = " << m << "\n" ;
18                 cout << "::m = " << ::m << "\n" ;
19
20         }
21
22                 cout << "\nEstamos fuera del bloque \n" ;
23                 cout << "m = " << m << "\n" ;
24                 cout << "::m = " << ::m << "\n" ;
25
26                 return 0;
27
28 }
```

Listing 23: Función sobrecargada en C++.

```
1  #include <iostream>
2
3  using namespace std;
4
5  int volumen(int);
6  double volumen(double,int);
7  long volumen(long,int,int);
8
9  int main(){
10
11         cout << volumne(10) << "\n";
12         cout << volumen(2.5,8) << "\n" ;
13         cout << volumen(100,75,15) << "\n" ;
14
15
16         return 0;
17
18 }
19
20 int volumen(int s){
21
```

```
22          return(s*s*s);
23
24  }
25
26
27  double volumen(double r, int h){
28
29  return(3.14519*r*r*h);
30
31  }
32
33
34  long volumen(long l, int b,int h){
35
36  return(l*b*h);
37
38  }
```

[1]

La clase es una forma de unir a los datos y a sus funciones asociadas para estar juntas. Esto permite que los datos y las funciones se oculten, si es necesario, de uso externo. Cuando se define una clase, estamos creando un nuevo tipo abstracto de datos que puede ser parecido a un tipo de datos creado. Generalmente, una especificación de clase tiene dos partes:

1. Declaración de clase
2. Definiciones de funciones de clase

Listing 24: Formato de clase en C++.

```
1  class Nombre{
2
3          private:
4                  Declacion de variables;
5                  Declaracion de funciones;
6
7          public:
8                  Declaracion de variables;
9                  Declaracion de funciones;
10
11 };
```

Ejemplo de una clase:

Listing 25: Ejemplo de clase en C++.

```
1  class elemento{
2
3
4                  int numero;
5                  float costo;
```

```
6
7           public:
8                   void getdata(int a, float b);
9                   void putdata(void);
10
11  };
```

Creación de objetos, se hace de la siguiente forma:

Listing 26: Creación de objetos en C++.

```
1  elemento x, y z;
```

Acceso a miembros de la clase:

Listing 27: Acceso a miembros de la clase en C++.

```
1  x.getdata(100,75.5);
```

[2]

Programación orientada a objetos

Algunos autores definen un objeto como un instancia de una clase, una instancia de un tipo de dato definido por el programador. Otros definen al objeto más ampliamente como uno u otro una instancia de una clase o una variable de un tipo constructor.

Interfaces

Es la que establece lo que se puede hacer para un objeto particular. Se declara una la clase **Lampara** de la siguiente forma:

1. Lampara
   - *a*) on()
   - *b*) off()
   - *c*) brillo()
   - *d*) atenuación()

Para crear el objeto **Lam1** se usa la siguiente sintaxis:

Listing 28: Declaración de objeto en java.

```
1 Lampara Lam1;
```

Para accesarse a la interface de la clase **Lampara** se usa la siguiente sintaxis:

Listing 29: Operación sobre un objeto en java.

```
1 Lam1.on();
```

Para poner los límites en una clase se usan las palabras clave: private, public, protected. Los objetos miembros de una clase son generalmente privados, haciendo de ellos inaccesibles a los clientes. La herencia es importante en programación orientada a objetos,

La clase **FiguraGeo**, tiene la siguiente composición:

1. FiguraGeo
   - *a*) dibujar()
   - *b*) borrar()

c) mover()
d) getColor()
e) setColor()

Se forman los objetos:
FiguraGeo circulo, cuadrado triangulo;
La clase termostato;

1. termostato

   a) bajarTemperatura()

La clase sistemaEnfriamento;

1. sistemaEnfriamento

   a) enfriar()

Listing 30: La clase doStuff en java.

```
1 void doStuff() {
2         s.borrar();
3         ...
4         s.dibujar();
5 }
```

El **ampersand** significa tomar la dirección del objeto que es pasado a la función (doStuff()).

Listing 31: Uso de la función doStuff en java.

```
1 circulo c;
2 triangulo t;
3 cuadrado cu;
4 doStuff(c);
5 doStuff(t);
6 doStuff(cu);
```

[4]

Referencia this

Cuando se llama una método instancia, esto se asocia con un objeto que lo llama. Si se tiene dos objetos que son instancias de la misma clase. Y se requiere que los dos objetos llamen al mismo método instancia, se hace por medio de la referencia **this**.

Listing 32: Uso en una función de la referncia this en java.

```
1 public void grow() {
2         this.weight +=
3         (0.01 * this.percentGrowthRate * this.weight);
4         this.age++;
5 } // end grow
```

[3]

Cuando trabajamos con constructores sobrecargados, es usual para un constructor invocar otro. En java, se hace usando otra forma de this. La forma general es la que sigue [Manual de java, pág]:

this(Lista de argumentos);

Listing 33: La clase MyClass en java.

```
1  class MyClass {
2          int a;
3          int b;
4          // initialize a and b individually
5          MyClass(int i, int j) {
6                  a = i;
7                  b = j;
8          }
9          // initialize a and b to the same value
10         MyClass(int i) {
11                 a = i;
12                 b = i;
13         }
14         // give a and b default values of 0
15         MyClass( ) {
16                 a = 0;
17                 b = 0;
18         }
19 }
```

Esta clase tiene tres constructores y cada uno inicializa los valores de a y b. Pero usando this() es posible reescribir MyClass como se muestra a continuación;

Listing 34: La clase MyClass en java.

```
1  class MyClass {
2          int a;
3          int b;
4          // initialize a and b individually
5          MyClass(int i, int j) {
6                  a = i;
7                  b = j;
8          }
9          // initialize a and b to the same value
10         MyClass(int i) {
11                 this(i, i); // invokes MyClass(i, i)
12         }
13         // give a and b default values of 0
14         MyClass( ) {
15                 this(0); // invokes MyClass(0)
16         }
17 }
```

Hay dos restricciones que se debe tener en cuenta cuando se usa this(): uno, no puede usar cualquier instancia del constructor de la clase en una llamada a this(). Dos, no se puede usar this() y super() en el mismo constructor.

Referencia instanceof [Manual java, pág 300]

A veces, es útil conocer el tipo de un objeto durante el tiempo de ejecución. Por ejemplo, podrías tener un hilo de ejecución que genera varios tipos de objetos y otro hilo que procesa estos objetos. En esta situación, puede ser útil que el subproceso de procesamiento para conocer el tipo de cada objeto cuando lo recibe. Otra situación en la que el conocimiento del tipo de un objeto en tiempo de ejecución es importante implica la conversión. En Java, una conversión no válida provoca un error de tiempo de ejecución. Muchos lanzamientos no válidos se pueden capturar en tiempo de compilación. Sin embargo, los moldes que involucran las jerarquías de clase pueden producir conversiones no válidas que solo se pueden detectar en tiempo de ejecución. Por ejemplo, una superclase llamada A puede producir dos subclases, llamadas B y C. Por lo tanto, lanzar un objeto B en el tipo A o lanzar un objeto C en el tipo A es legal, pero lanzar un objeto B en el tipo C (o viceversa) no es legal. Debido a que un objeto de tipo A puede referirse a objetos de B o C, cómo ¿Puede saber, en tiempo de ejecución, a qué tipo de objeto se hace referencia antes de intentar el elenco para escribir C? Podría ser un objeto de tipo A, B o C. Si es un objeto de tipo B, una ejecución se lanzará una excepción de tiempo. Java proporciona la instancia del operador en tiempo de ejecución para responder esta pregunta.

La forma general de instanceof:

Listing 35: Sintaxis de la instanceOf en java.

```
1 referenciaObjeto instanceof tipo;
```

Aquí, objref es una referencia a una instancia de una clase, y type es un tipo de clase. Si objref es del tipo especificado o se puede convertir en el tipo especificado, entonces el operador instanceof se evalúa como cierto. De lo contrario, su resultado es falso. Por lo tanto, instanceof es el medio por el cual su programa puede obtener información de tipo de tiempo de ejecución sobre un objeto. El siguiente programa demuestra instanceof:

Listing 36: demistración de instanceOf en java.

```
1  // Demostracion del operador instanceof.
2  class A {
3          int i, j;
4  }
5  class B {
6          int i, j;
7  }
8  class C extends A {
9          int k;
10 }
11 class D extends A {
```

```
12          int k;
13  }
14  class InstanceOf {
15          public static void main(String args[]) {
16                  A a = new A();
17                  B b = new B();
18                  C c = new C();
19                  D d = new D();
20
21                  if(a instanceof A)
22                  System.out.println("a
23  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if(b␣instanceof␣B)
24  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣System.out.println("b
25                  if(c instanceof C)
26                  System.out.println("c
27  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if(c␣instanceof␣A)
28  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣System.out.println("c
29                  I/O , Applets, and Other Topics
30                  is instance of A");
31  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣is␣instance␣of␣B");
32                  is instance of C");
33  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣can␣be␣cast␣to␣A");
34                  if(a instanceof C)
35                  System.out.println("a␣can␣be␣cast␣to␣C");
36                  System.out.println();
37                  // compare types of derived types
38                  A ob;
39                  ob = d; // A reference to d
40                  System.out.println("ob␣now␣refers␣to␣d");
41                  if(ob instanceof D)
42                  System.out.println("ob␣is␣instance␣of␣D");
43                  System.out.println();
44                  ob = c; // A reference to c
45                  System.out.println("ob␣now␣refers␣to␣c");
46                  if(ob instanceof D)
47                  System.out.println("ob␣can␣be␣cast␣to␣D");
48                  else
49                  System.out.println("ob␣cannot␣be␣cast␣to␣D");
50                  if(ob instanceof A)
51                  System.out.println("ob␣can␣be␣cast␣to␣A");
52                  System.out.println();
53                  // all objects can be cast to Object
54                  if(a instanceof Object)
55                  System.out.println("a␣may␣be␣cast␣to
56  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if(b␣instanceof␣Object)
57  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣System.out.println("b may be cast to
58                  if(c instanceof Object)
59                  System.out.println("c␣may␣be␣cast␣to
60  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if(d␣instanceof␣Object)
61  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣System.out.println("d may be cast to
```

```
62          }
63  }
```

La mayoría de los programas no necesitan la instancia del operador porque, en general, usted sabe el tipo de objeto con el que está trabajando. Sin embargo, puede ser muy útil cuando estás escribiendo rutinas generalizadas que operan en objetos de una jerarquía de clases compleja.

Del libro de Dean [pág. 522] instanceof Como ha visto, cada vez que una referencia genérica llama a un método polimórfico, la JVM utiliza el tipo de referencia. Objeto creado para decidir a qué método llamar. Es posible que desee hacer algo similar explícitamente en su código. En particular, es posible que desee ver si un objeto referenciado es una instancia de alguna clase en particular. Tu puedes hacer esto con un operador especial llamado el operador instanceof (tenga en cuenta que la .º.en instanciaof es menor caso). Usando el ejemplo Mascotas nuevamente, suponga que desea imprimir "Wags tail"si el objeto obj es una instancia de clase Perro o cualquier clase descendiente de la clase Perro. Puede hacerlo con la instrucción if en la parte inferior del método principal en la figura 13.8. Por lo tanto, el operador instanceof proporciona una forma simple y directa de ordenar los distintos tipos de objetos a los que una variable de referencia genérica puede hacer referencia.

Listing 37: Uso de la palabra instanceOf en java.

```
1  import java.util.Scanner;
2  public class Pets2 {
3          public static void main(String[] args) {
4                  Scanner stdIn = new Scanner(System.in);
5                  Object obj;
6                  System.out.print("Que␣tipo␣de␣mascota
7  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣prefieres?\n" +
8                  "Escribir␣d␣para␣perro␣or␣c␣para␣gatos:");
9                  if (stdIn.next().equals("d"))          {
10                         obj = new Dog();
11                 }
12                 else {
13                         obj = new Cat();
14                 }
15
16                 if (obj instanceof Dog) {
17                         System.out.println("Menear␣la
18 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣cola");
19                 }
20         } // Fin de main
21 } // Fin de clase Pets2
```

Clases múltiples

Cuando se usan dos clases o más, las clases se reconocen de dos maneras: una clase maneja dora que contiene al método main y otra clase manejada que contiene algunos métodos. Un ejemplo de esta clase es un sistema para abrir la puerta de un estacionamiento.

[3]

Métodos

La sintaxis para definir un método es la siguiente:

Listing 38: Sintaxis para definir un método en java.

```
modifier returnValueType methodName(list of parameters) {
// Method body;
}
```

El siguiente es un programa que usa un método:

Listing 39: Programa con un método en java.

```
public class TestMax {
        /** Main method */

        public static void main(String[] args) {

                int i = 5;
                int j = 2;
                int k = max(i, j) ;

                System.out.println("The maximum between "
                 + i + " and " + j + " is " + k);

        }

        /** Return the max between two numbers */
        public static int max(int num1, int num2) {

                int result;

                if (num1 > num2)
                        result = num1;

                else
                        result = num2;

                return result;
```

```
27          }
28  }
```

Abstracción de datos

Métodos heredados. Varias convenciones de Java permiten que un tipo de datos aproveche mecanismos de lenguaje integrados al incluir métodos específicos en la API. Por ejemplo, todos los tipos de datos de Java heredan un método toString () que devuelve una representación de cadena de los valores de tipo de datos. Java llama a este método cuando cualquier valor de tipo de datos debe ser concatenado, revestido con un valor de cadena con el operador +. La implementación predeterminada no es particularmente útil (da una representación de cadena de la dirección de memoria del tipo de datos valor), por lo que a menudo ofrecemos una implementación que anula el valor predeterminado e incluimos toString () en la API cada vez que lo hacemos. Otros ejemplos de tales métodos incluyen equals (), compareTo () y hashCode () (consulte la página 101).

Objetos. Naturalmente, puede declarar que una cabeza variable se asociará con datos de tipo Contador con el código Contra-cabezas; pero, ¿cómo puede asignar valores o especificar operaciones? La respuesta a esta pregunta implica un concepto fundamental en la abstracción de datos: un objeto es una entidad que puede asumir. Un valor de tipo de datos. Los objetos se caracterizan por tres propiedades esenciales: estado, identidad y comportamiento. El estado de un objeto es un valor de su tipo de datos. La identidad de un objeto distingue un objeto de otro. Es útil pensar en la identidad de un objeto como el lugar donde su valor se almacena en la memoria. El comportamiento de un objeto es el efecto de las operaciones de tipo de datos. La implementación tiene la responsabilidad exclusiva de mantener la identidad de un objeto, de modo que el código del cliente Identity puede usar un tipo de datos sin tener en cuenta la representación de su estado al ajustarse a una API que describe el comportamiento de un objeto. El estado de un objeto podría usarse para proporcionar información a un cliente o causar un efecto secundario o ser cambiado por una de las operaciones de su tipo de datos, pero los detalles de la representación del valor del tipo de datos no son relevantes para el código del cliente. Una referencia es un mecanismo para acceder a un objeto. La nomenclatura de Java deja en claro la distinción de dos primitivos objetos tipos Counter (donde las variables están asociadas con valores) usando el término tipos de referencia para tipos no primitivos. Los detalles de implementación las referencias varían en las implementaciones de Java, pero es útil pensar en una referencia como una dirección de memoria, como se muestra a la derecha (por brevedad, usamos direcciones de memoria de tres dígitos en el diagrama). Creando objetos. Cada valor de tipo de datos se almacena en un objeto. A crear (o instanciar) un objeto individual, invocamos un constructor utilizando la palabra clave **new**, seguida del nombre de la clase, seguido de () (o una lista de valores de argumentos encerrados entre paréntesis, si el constructor toma argumentos). Un constructor no tiene tipo de retorno porque siempre devuelve una referencia a un objeto de su tipo de datos. Cada vez que un cliente usa new (), el sistema:

* Asigna espacio de memoria para el objeto Representación de objetos * In-

voca al constructor para inicializar su valor. * Devuelve una referencia al objeto En el código del cliente, generalmente creamos objetos en una declaración de inicialización que asocia un variable con el objeto, como a menudo hacemos con variables de tipos primitivos. A diferencia de los tipos primitivos, las variables están asociadas a referencias a objetos, no a los valores de tipo de datos en sí. Podemos crear cualquier cantidad de objetos de la misma clase cada objeto tiene su propia identidad y puede o no almacenar lo mismo valor como otro objeto del mismo tipo. Por ejemplo, el código

Counter heads = new Counter("heads"); Counter tails = new Counter("tails");

crea dos objetos de contador diferentes. En un tipo de datos abstracto, detalles de la representación La porción del valor está oculta del código del cliente. Puede suponer que el valor asociado junto con cada objeto Counter hay un nombre de cadena y una lista, pero no puede escribir código que depende de cualquier representación específica (o incluso saber si esa suposición es cierto, tal vez la cuenta sea un valor largo).

Variables de instancia. Para definir el tipo de datos valores (el estado de cada objeto), declaramos variables de instancia de la misma manera que declaramos variables locales. Hay un Las variables de instancia en ADT son privadas distinción crítica entre instancia variables y las variables locales dentro de una estática método o un bloque al que está acostumbrado: solo hay un valor correspondiente a cada variable local en un momento dado, pero hay numerosos valores correspondientes a cada variable de instancia (una para cada objeto que es una instancia del tipo de datos). No hay ambigüedad con esta disposición, porque cada vez que accedemos a una variable de instancia, lo hacemos con un nombre de objeto: ese objeto es aquel cuyo valor estamos accediendo. Además, cada declaración está calificada por un modificador de visibilidad. En implementaciones de ADT, nosotros usar privado, usando un lenguaje Java mechansim para hacer cumplir la idea de que el representante La porción de un ADT debe ocultarse al cliente, y también final, si el valor no debe ser cambiado una vez que se inicializa. El contador tiene dos variables de instancia: un nombre de valor de cadena y un recuento de valor int. Si tuviéramos que usar variables de instancia pública (permitido en Java) el tipo de datos, por definición, no sería abstracto, por lo que no lo hacemos.

Constructores. Cada clase Java tiene al menos un constructor que establece un objeto identidad. Un constructor es como un método estático, pero puede referirse directamente a varianza de instancia Ables y no tiene valor de retorno. Generalmente, el propósito de un constructor es inicializar Las variables de instancia. Cada constructor crea un objeto y proporciona al cliente un referencia a ese objeto. Los constructores siempre comparten el mismo nombre que la clase. Podemos sobrecargar el nombre y tener múltiples constructores con diferentes firmas, tal como con métodos Si no se define ningún otro constructor, un constructor predeterminado sin argumentos es implícito, no tiene argumentos e inicializa los valores de instancia a los valores predeterminados. El valor por defecto los valores de las variables de instancia son 0 para tipos numéricos primitivos, falso para booleano y nulo para los tipos de referencia. Estos valores predeterminados se pueden cambiar utilizando declaraciones de inicialización para variables de instancia. Java invoca automáticamente un constructor cuando un programa cliente usa la palabra clave new. Los constructores sobrecargados se usan generalmente para inicializar variables de instancia a valores proporcionados por el cliente que no sean los predeterminados. Por ejemplo,

Counter tiene un constructor de un argumento que inicializa la variable de instancia de nombre al valor dado como argumento (dejando que la variable de instancia de conteo se inicialice al valor predeterminado 0).

[8, pág 60-111]

0.0.13. Uso de la palabra clave this

A veces, un método deberá referirse al objeto que lo invocó. Para permitir esto, Java define la esta palabra clave **this** se puede utilizar dentro de cualquier método para referirse al objeto actual. Es decir, **this** siempre es una referencia al objeto en el que se invocó el método. Puedes usar **this** en cualquier lugar esto permite una referencia a un objeto del tipo de clase actual. Para comprender mejor a qué se refiere esto, considere la siguiente versión de Box():

0.0.14. Constructor

Puede ser tedioso inicializar todas las variables en una clase cada vez que se crea una instancia. Incluso cuando agrega funciones convenientes como setDim (), sería más simple y conciso para hacer toda la configuración en el momento en que se crea el objeto por primera vez. Porque el requisito para la inicialización es tan común, Java permite que los objetos se inicialicen cuando se están creando. Esta inicialización automática se realiza mediante el uso de un constructor. Un constructor inicializa un objeto inmediatamente después de la creación. Tiene el mismo nombre que la clase en la que reside y es sintácticamente similar a un método. Una vez definido, el constructor se llama automáticamente inmediatamente después de crear el objeto, antes de que se complete el nuevo operador. Los constructores se ven un poco extraños porque no tienen tipo de retorno, ni siquiera nulos. Esto es porque el tipo de retorno implícito del constructor de una clase es el tipo de clase en sí. Es el constructor trabajo para inicializar el estado interno de un objeto para que el código que crea una instancia tenga un objeto totalmente inicializado y utilizable de inmediato.

Puede volver a trabajar el ejemplo de Box para que las dimensiones de un box sean automáticamente inicializado cuando se construye un objeto. Para hacerlo, reemplace setDim() con un constructor. Comencemos definiendo un constructor simple que simplemente establece las dimensiones de cada cuadro a los mismos valores. Esta versión se muestra aquí:

0.0.15. Constructor parametrizado

Si bien el constructor Box () en el ejemplo anterior inicializa un objeto Box, no es muy útil: todas las cajas tienen las mismas dimensiones. Lo que se necesita es una forma de construir objetos Box de varias dimensiones. La solución fácil es agregar parámetros al constructor. Como probablemente puedas adivinar, esto los hace mucho más útiles. Por ejemplo, la siguiente versión de Box define un constructor parametrizado que establece las dimensiones de un cuadro según lo especificado para esos parámetros. Presta especial atención a cómo se crean los objetos Box.

0.0.16. Interfaces

Usando la palabra clave interface, puede abstraer completamente la interfaz de una clase desde su implementación. Es decir, utilizando la interfaz, puedes especificar qué debe hacer una clase, pero no cómo lo hace. Interfaces son sintácticamente similares a las clases, pero carecen de variables de instancia y sus métodos son declarados sin ningún cuerpo. En la práctica, esto significa que puedes definir interfaces que no hacen suposiciones sobre cómo se implementan. Una vez que se define, cualquier número de las clases pueden implementar una interfaz. Además, una clase puede implementar cualquier número de interfaces. Para implementar una interfaz, una clase debe crear el conjunto completo de métodos definidos por la interfaz. Sin embargo, cada clase es libre de determinar los detalles de su propia implementación. Al proporcionar la palabra clave de **interface**, Java permite utilizar completamente la ïnterfaz única, aspecto de métodos múltiples"del polimorfismo.

Las interfaces están diseñadas para admitir la resolución dinámica de métodos en tiempo de ejecución. Normalmente, para que un método se llame de una clase a otra, ambas clases deben estar presentes en tiempo de compilación para que el compilador de Java pueda verificar que las firmas del método sean compatibles. Este requisito por sí solo hace una clasificación estática y no extensible de ambiente. Inevitablemente en un sistema como este, la funcionalidad se eleva cada vez más en la jerarquía de clases para que los mecanismos estén disponibles para más y más subclases. Las interfaces están diseñadas para evitar este problema. Desconectan la definición de un método o conjunto de métodos de la jerarquía de herencia. Dado que las interfaces están en una jerarquía diferente de clases, es posible que las clases que no están relacionadas en términos de la jerarquía de clases implementen la misma interfaz. Aquí es donde se realiza el verdadero poder de las interfaces.

RECORDAR Cuando usted implemente un método interface. usted debe declararlo como público.

Listing 40: Formato para interface en java.

```
acceso interface nameVariable {
        return-type method-name1(parameter-list);
        return-type method-name2(parameter-list);
        type final-varname1 = value;
        type final-varname2 = value;
        // ...
        return-type method-nameN(parameter-list);
        type final-varnameN = value;
}
```

[7, pág 119]

Superclases y subclases La programación orientada a objetos permite manipular clases nuevas, de clases que ya existen, esto es llamado **herencia**. La **herencia** es una importante y poderosa característica de java para reusar software.

Se usa una clase para modelar objetos del mismo tipo. Las clases diferentes pueden tener características y comportamientos comunes, lo cual puede ser generalizado en una clase que es compartida por otra clase.

Ejemplo Considerar objetos geométricos. Supón que quieres modelar objetos geométricos tales como círculos y rectángulos. Los objetos geométricos tienen características y comportamientos comunes. Se pueden dibujar con ciertos colores, sólidos y huecos.

Una clase C_1 es una clase extendida de otra clase C_2 es llamada una subclase, y C_2 es una superclase. Una superclase es llamada también clase padre, o clase base, y una subclase es una clase hija, o clase extendida, o clase derivada.

Clases abstractas ¿Como puedo escribir código para responder a la acción de un usuario? Para escribir ese código, tienes que saber **interfaces**.

Una **interface** es para definir el comportamiento físico de las clases (especialmente clases no relacionadas). Algo relacionado es **objetivo:clase abstracta**.

En la herencia jerárquica, las clases son más especificas y concretas con cada nueva subclase. Si se quiere mover una subclase de regreso a una superclase, la clase es más general y menos especifica. El diseño de clases debe asegurar que una superclase contenga características comunes de su subclase. Algunas veces una superclase es tan abstracta que no puede tener cualquier instancia especifica. Tal clase, se llama clase abstracta.

En el Capítulo 11, GeometricObject se definió como la superclase para Circle y Rectángulo, GeometricObject modela las características comunes de los objetos geométricos. Ambos Circle y Rectangle contienen los métodos getArea() y getPerimeter() para calcular el área y el perímetro de un círculo y un rectángulo. De manera que puedes calcular áreas y perímetros para todos los objetos geométricos, es mejor definir los métodos getArea() y getPerimeter() en la clase GeometricObject. Sin embargo, estos métodos no pueden implementarse en la clase GeometricObject, porque su implementación depende del tipo específico de objeto geométrico. Dichos métodos se denominan métodos abstractos y se denotan usando el modificador **abstract** en el encabezado del método. Después de definir los métodos en GeometricObject, se convierte en una clase abstracta. Las clases abstractas se denotan usando el modificador **abstract** en el encabezado de la clase. En notación gráfica UML, los nombres de las clases abstractas y sus métodos abstractos están en cursiva, como se muestra en la Figura 14.2. El listado 41 da el código fuente de la nueva clase GeometricObject.

Listing 41: Clase abstracta en java.

```
1  public abstract class GeometricObject {
2          private String color = "white";
3          private boolean filled;
4          private java.util.Date dateCreated;
5
6          /** Construct por default de un objeto geometri */
7          protected GeometricObject() {
8                  dateCreated = new java.util.Date();
9          }
10
11 /** Construct un objeto geometric con color y valor
12  solido */
13 protected GeometricObject(String color, boolean filled) {
14
15         dateCreated = new java.util.Date();
```

```
16          this.color = color;
17          this.filled = filled;
18          }
19
20          /** Return color */
21          public String getColor() {
22          return color;
23          }
24
25          /** Set a new color */
26          public void setColor(String color) {
27          this.color = color;
28          }
29
30          /** Return filled. Since filled is boolean,
31          * the get method is named isFilled */
32          public boolean isFilled() {
33          return filled;
34          }
35
36          /** Set a new filled */
37          public void setFilled(boolean filled) {
38                  this.filled = filled;
39          }
40          /** Get dateCreated */
41          public java.util.Date getDateCreated() {
42                  return dateCreated;
43          }
44          /** Return a string representation of this
45           object */
46          public String toString() {
47                  return "created on " + dateCreated
48                   + "\ncolor: " + color +
49                  " and filled: " + filled;
50          }
51          /** Abstract method getArea */
52          public abstract double getArea();
53                  /** Abstract method getPerimeter */
54                  public abstract double getPerimeter();
55          }
```

La clase abstracta GeometricObject define las características comunes (datos y métodos) para objetos geométricos y proporciona constructores adecuados. Porque no sabes cómo calcular áreas y perímetros de objetos geométricos, getArea y getPerimeter son definidos como métodos abstractos. Estos métodos se implementan en las subclases, la implementación de Circle and Rectangle es la misma que en los listados 14.2 y 14.3, excepto que extienden la clase GeometricObject definida en este capítulo, de la siguiente manera:

Listing 42: Subclase Circle en java.

```
1  public class Circle extends GeometricObject {
2
3          // Same as lines 2-46 in Listing 11.2, so omitted
4  }
```

Listing 43: Subclase Rectangle en java.

```
1  public class Rectangle extends GeometricObject {
2
3          // Same as lines 2-49 in Listing 11.3, so omitted
4  }
```

Listing 44: Superclase en java.

```
1  public class GeometricObject1 {
2
3          private String color = "white";
4          private boolean filled;
5          private java.util.Date dateCreated;
6
7          /** Construct a default geometric object */
8
9          public GeometricObject1() {
10                 dateCreated = new java.util.Date();
11
12         }
13
14         /** Construct a geometric object with the
15          specified color
16
17         * and filled value */
18
19         public GeometricObject1(String Color, boolean
20                  filled) {
21                 dateCreated = new java.util.Date();
22
23                 this.color = color;
24                 this.filled = filled;
25
26         }
27
28         /** Return color */
29
30         public String getColor() {
31
32                 return color;
33
34         }
35         /** Set a new color */
36         public void setColor(String color) {
```

```
37                 this.color = color;
38         }
39         /** Return filled. Since filled is boolean,
40         its get method is named isFilled */
41         public boolean isFilled() {
42                 return filled;
43         }
44         /** Set a new filled */
45         public void setFilled(boolean filled) {
46                 this.filled = filled;
47         }
48         /** Get dateCreated */
49         public java.util.Date getDateCreated() {
50                 return dateCreated;
51         }
52         /** Return a string representation of this
53          object */
54         public String toString() {
55                 return "created on " + dateCreated +
56                  "\ncolor: " + color +
57                 " and filled: " + filled;
58         }
59 }
```

Listing 45: Subclase Circle4 en java.

```
1  public class Circle4 extends GeometricObject1 {
2
3  private double radius;
4
5
6  public Circle4() {
7
8  }
9
10 public Circle4(double radius) {
11
12 this.radius = radius;
13
14 }
15
16
17 public Circle4(double radius, String color, boolean
18          filled) {
19
20         this.radius = radius;
21         setColor(color);
22         setFilled(filled);
23 }
24
```

```
25
26 /** Return radius */
27 public double getRadius() {
28
29         return radius;
30 }
31
32 /** Set a new radius */
33 public void setRadius(double radius) {
34
35         this.radius = radius;
36
37 }
38
39 /** Return area */
40 public double getArea() {
41
42         return radius * radius * Math.PI;
43 }
44 /** Return diameter */
45 public double getDiameter() {
46         return 2 * radius;
47 }
48 /** Return perimeter */
49 public double getPerimeter() {
50         return 2 * radius * Math.PI;
51 }
52 /* Print the circle info */
53 public void printCircle() {
54         System.out.println("The circle is created " +
55         getDateCreated() +
56         " and the radius is " + radius);
57 }
58 }
```

[6, cap-11]

Interfaces de otro libro Una interface es una clase parecida a un constructor que contiene solamente constantes y métodos abstractos. En muchas formas una interface es similar a una clase abstracta, pero su intento es para especificar comportamientos físicos comunes de objetos. Por ejemplo, usando interfaces apropiadas, usted puede especificar que los objetos son comparables, editables, y/o clonables.

Para distinguir una clase de una interface, java usa la siguiente sintaxis para definir una interface:

Listing 46: Sintaxis de interface en java.

```
1 modifier interface InterfaceName {
2         /** Constant declarations */
3         /** Method signatures */
```

```
4 }
```

Una interface es parecida a una clase especial en java. Cada interface es compilada en un archivo separado de Bytecode, parecido a una clase regular. Como con una clase abstracta, usted no puede crear una instancia de una interface que esta usando el operador **new**, pero en muchos casos usted puede usar una interface más o menos de la misma forma que una clase abstracta. Por ejemplo, usted puede usar una interface como un tipo de dato para una variable referencia, como un **cast** y así.

Ahora puede usar la interfaz Edible para especificar si un objeto es editable. Esto es logrado al permitir que la clase para el objeto implemente esta interfaz usando la palabra clave **implements**. Por ejemplo, las clases Pollo y Fruta en el Listado 14.6 (líneas 14, 23) implementan la interfaz editable. La relación entre la clase y la interfaz, se conoce como herencia de interfaz. Como la herencia de interfaz y la herencia de clase son esenciales de manera similar, simplemente nos referiremos a ambos como herencia.

Ejemplo de interface Se quiere diseñar un método genérico para encontrar el objeto más grande del mismo tipo, tal como dos estudiantes, dos datos, dos círculos, dos rectángulos, dos cuadrados. Para completar esto, los objetos deben ser comparables. Java tiene la interfaz Comparable, la interfaz se define como:

Listing 47: Ejemplo de interface en java.

```
1 // Interface para comparar objetos, definida en java.lang
2
3 package java.lang;
4
5 public interface Comparable {
6         public int compareTo(Object o);
7 }
```

El método compareTo determina el orden del objeto con el objeto especificado o y regresa un entero negativo, cero, o entero positivo si este objeto es menor que, igual a, o más grande que o.

[6, 458]

Método abstracto ¿Porque un método abstracto? Se maravilla con la declaración de getArea y getPerimeter como abstract en la clase GeometricObject en lugar de definir-los en cada subclase. El siguiente ejemplo muestra los beneficios de definirlos en la clase GeometricObject. El ejemplo siguiente crea dos objetos geométricos, un círculo y un rectángulo, invoca el método equalArea para verificar si tienen áreas iguales e invoca el método displayGeometricObject para visualizar los objetos.

Listing 48: Ejemplo de métodos abstractos en java.

```
1 public class TestGeometricObject {
2         /** Main method */
3         public static void main(String[] args) {
4
5                 // Create two geometric objects
```

```
6                   GeometricObject geoObject1 = new
7                    Circle(5);
8                   GeometricObject geoObject2 = new
9                    Rectangle(5, 3);
10
11                  System.out.println(
12                  "The two objects have the same area? " +
13                  equalArea(geoObject1, geoObject2) );
14
15                  // Display circle
16                  displayGeometricObject(geoObject1);
17
18                  // Display rectangle
19                  displayGeometricObject(geoObject2);
20          }
21
22          /** A method for comparing the areas of two
23           geometric objects */
24          public static boolean equalArea(
25                  GeometricObject object1,
26                  GeometricObject object2) {
27
28                          return object1.getArea() ==
29                           object2.getArea();
30                  }
31
32                  /** A method for displaying a geometric
33                   object */
34          public static void displayGeometricObject(
35                  GeometricObject object) {
36
37                  System.out.println();
38                  System.out.println("The area is " +
39                   object.getArea());
40                  System.out.println("The perimeter is " +
41                   object.getPerimeter());
42          }
43  }
```

Resumen de abstracción método abstracto en clase abstracta - Un método abstracto no puede estar contenido en una clase no abstracta. Si una subclase en resumen, la subclase debe estar definida en resumen. En otras palabras, en una subclase no abstracta extendida desde una clase abstracta, todos los Se deben implementar métodos abstractos. También tenga en cuenta que los métodos abstractos no son estáticos. el objeto no se puede crear desde clase abstracta

la superclase no implementa todos los métodos abstractos - No se puede crear una instancia de una clase abstracta utilizando el nuevo operador, pero aún puede define sus constructores, que se invocan en los constructores de sus subclases. por Por ejemplo, los constructores de GeometricObject se invocan en

la clase Circle y la clase Rectángulo.

- Una clase que contiene métodos abstractos debe ser abstracta. Sin embargo, es posible definir una clase abstracta que no contiene métodos abstractos. En este caso, no puedes crear instancias de la clase utilizando el nuevo operador. Esta clase se usa como clase base para definir una nueva subclase.

superclase de clase abstracta puede ser concreto - Una subclase puede ser abstracta incluso si su superclase es concreta. Por ejemplo, el objeto La clase es concreta, pero sus subclases, como GeometricObject, pueden ser abstractas.

método concreto anulado ser abstracto - Una subclase puede anular un método de su superclase para definirlo como abstracto. Esto es muy inusual pero es útil cuando la implementación del método en la superclase se vuelve inválido en la subclase. En este caso, la subclase debe definirse como abstracta.

clase abstracta como tipo - No puede crear una instancia a partir de una clase abstracta utilizando el nuevo operador, sino un La clase abstracta se puede utilizar como un tipo de datos. Por lo tanto, la siguiente declaración, que crea una matriz cuyos elementos son del tipo GeometricObject, es correcta. clase abstracta sin resumen método Objetos GeometricObject [] = new GeometricObject [10]; Luego puede crear una instancia de GeometricObject y asignar su referencia a la matriz como esta:

Programas de interfaces Se crea la interfaz en el programa **Commission.java**, se crea una clase **Commissioned.java**, después se crea la clase en el programa **SalariedAndCommissioned.java** y todo esto se corre en el programa **Payroll3.java**.

Listing 49: Ejemplo de interface en java.

```
1  /**************************
2  * Commission.java
3  * Dean & Dean
4  *
5  * Esta inteface especifica un atributo comun
6  * y declara un comportamiento comun de empleados
7   comisionados.
8  **************************/
9  interface Commission {
10         double COMMISSION_RATE = 0.10;
11         void addSales(double sales);
12 } // fin interface Commission
```

Listing 50: Ejemplo de clase Employee2 en java.

```
1  /**********************************
2  * Employee2.java
3  * Dean & Dean
4  *
5  * Esta clase abstract describe a employees.
6  **********************************/
7
8  public abstract class Employee2 {
9          public abstract double getPay();
```

```
10          private String name;
11
12  //************************************
13          public Employee2(String name) {
14                  this.name = name;
15          }
16  //************************************
17          public void printPay(int date) {
18                  System.out.printf("%2d %10s: $%8.2f\n",
19                  date, name, getPay());
20          } // fin de printPay
21  } // fin de la clase Employee2
```

Listing 51: Ejemplo de uso de interface en java.

```
1   /*******************************
2   * Commissioned.java
3   * Dean & Dean
4   *
5   * Esta clase representa empleados en comision commission.
6   *******************************/
7
8   public class Commissioned extends Employee2
9    implements Commission {
10          private double sales = 0.0;
11  //*********************************
12          public Commissioned(String name) {
13                  super(name);
14                  this.sales = sales;
15          } // fin del constructor
16  //*********************************
17          public void addSales(double sales) {
18                  this.sales += sales;
19          } // fin de addSales
20
21  //************************************
22          public double getPay() {
23                  double pay = COMMISSION_RATE * sales;
24                  sales = 0.0;
25                  return pay;
26          } // fin de getPay
27  } // fin de la clase Commissioned
```

Listing 52: Ejemplo de clases múltiple en java.

```
1   /*******************************
2   * SalariedAndCommissioned.java
3   * Dean & Dean
4   *
5   * Esta clase representa salario y comision de empleados.
```

```
6  ********************************/
7  public class SalariedAndCommissioned extends
8   Salaried2 implements Commission {
9          private double sales;
10 //*******************************
11         public SalariedAndCommissioned(String name,
12          double salary) {
13                 super(name, salary);
14         } // fin de constructor
15 //*******************************
16         public void addSales(double sales) {
17                 this.sales += sales;
18         } // fin de addSales
19
20 //*******************************
21         public double getPay() {
22                 double pay =
23                 super.getPay() + COMMISSION_RATE * sales;
24                 sales = 0.0; // reinicializar para
25                  el siguiente periodo de pago
26                 return pay;
27         } // fin de getPay
28 } // fin de clase SalariedAndCommissioned
```

Listing 53: Ejemplo de interface en java.

```
1  /*******************************
2  * Payroll3.java
3  * Dean & Dean
4  *
5  * Esta clase hires y paga cuatro diferentes tipos de
6   empleos.
7  ********************************/
8  public class Payroll3 {
9     public static void main(String[] args) {
10    Employee2[] employees = new Employee2[100];
11    Hourly2 hourly;
12    employees[0] = new Hourly2("Anna", 25.0);
13    employees[1] = new Salaried2("Simon", 48000);
14    employees[2] = new Hourly2("Donovan", 20.0);
15    employees[3] = new Commissioned("Glen");
16    employees[4] = new SalariedAndCommissioned("Carol",
17   24000);
18    ((Commission) employees[3]).addSales(15000);
19    ((Commission) employees[4]).addSales(15000);
20
21    // Esto arbitrariamente asume que el mes de payroll's
22    // comienza el lunes (day = 2), y contiene 30 days.
23    for (int date=1,day=2; date<=15; date++,day++,day%=7) {
24       for (int i=0;
```

```
25          i<employees.length && employees[i] != null; i++)
26      {
27           if (day > 0 && day < 6
28          && employees[i] instanceof Hourly2)
29      {
30      hourly = (Hourly2) employees[i];
31      hourly.addHours(8);
32  }
33      if ((day == 5 && employees[i] instanceof Hourly2) ||
34      (date%15 == 0 &&
35      (employees[i] instanceof Salaried2 ||
36      employees[i] instanceof Commissioned)))
37      {
38          employees[i].printPay(date);
39      }
40  } // fin para i
41  } // fin para date
42  } // fin de main
43  } // fin de la clase Payroll3
```

[3, págs, 536-538]

Conceptos básicos del java Como se forma una clase, que se presenta en el siguiente código:

Listing 54: Ejemplo de clase en java.

```
1  class thermostat {
2          private float currentTemp();
3          private float desiredTemp();
4          public void furnace_on() {
5          // el cuerpo del metodo va aqui
6          }
7          public void furnace_off() {
8          // el cuerpo del metodo va aqui
9          }
10 } // fin de la clase thermostat
```

Interface en java Hemos visto cómo un programa se puede dividir en clases separadas. ¿Cómo funcionan estas clases? ¿Interactuar la una con la otra? La comunicación entre clases y la división de responsabilidad entre ellas, son aspectos importantes de la programación orientada a objetos. Esto es especialmente cierto cuando una clase puede tener muchos usuarios diferentes. Típicamente una clase puede ser utilizada una y otra vez por diferentes usuarios (o el mismo usuario) para diferentes propósitos, por ejemplo, es posible que alguien pueda usar la clase LowArray en algún otro programa para almacenar los números de serie de sus cheques de viajero. La clase puede manejar esto tan bien como almacenar el número de jugadores de béisbol. Si una clase es utilizada por muchos programadores diferentes, la clase debe diseñarse de modo que sea fácil de usar. La forma en que un usuario de clase se relaciona con la clase se denomina

interfaz de clase. Debido a que los campos de clase suelen ser privados, cuando hablamos de la interfaz, generalmente significa que los métodos de clase: qué hacen y cuáles son sus argumentos. Se llama a estos métodos en los que un usuario de clase interactúa con un objeto de la clase. Una de las ventajas importantes conferidas por la programación orientada a objetos es que una interfaz de clase puede ser diseñado para ser lo más conveniente y eficiente posible. La figura 2.3 es una fantasía interpretar de la interfaz LowArray.

[5, pág,16]

Listing 55: Ejemplo de clase lowArray en java.

```
1  // lowArray.java
2  // se muestra la clase array con interface de low-level
3  // para correr el programa: C>java LowArrayApp
4  import java.io.*; // para la I/O
5  ////////////////////////////////
6  class LowArray {
7          private double[] a; // referencia a un array a
8
9          // constructor
10         public LowArray(int size){
11         a = new double[size];
12         }
13
14         // put elemento dentro de array
15         public void setElem(int index, double value) {
16                 a[index] = value;
17         }
18
19         // get elemento de array
20         public double getElem(int index) {
21         return a[index];
22         }
23 } // fin de la clase LowArray
24
25 ////////////////////////////////
26 class LowArrayApp {
27         public static void main(String[] args){
28                 LowArray arr; //referencia
29                 arr = new LowArray(100); //crear el
30                  objeto LowArray
31                 int nElems = 0; //numero de elementos
32                  en array
33                 int j; // variable del ciclo
34
35                 arr.setElem(0, 77); // insert 10 items
36                 arr.setElem(1, 99);
37
38 arr.setElem(2,44);
39 arr.setElem(3,55);
40 arr.setElem(4,22);
```

```
41 arr.setElem(5,88);
42 arr.setElem(6,11);
43 arr.setElem(7,00);
44 arr.setElem(8,66);
45 arr.setElem(9,33);
46 nElems = 10; // ahora 10 elementos en array
47
48 //--------------------------------------
49
50 for(j=0; j<nElems; j++) // visualizar los elementos
51         System.out.print(arr.getElem(j) + " ");
52 System.out.println("");
53 //--------------------------------------
54 -
55 int searchKey = 26; // busqueda de elementos data
56 for(j=0; j<nElems; j++) // para cada elemento,
57         if(arr.getElem(j) == searchKey) // encuentra el
58          elemento?
59                 break;
60 if(j == nElems) // no
61         System.out.println("Can't find " + searchKey);
62 else // yes
63         System.out.println("Found " + searchKey);
64 //--------------------------------------
65
66 // borrar 55 valores
67 for(j=0; j<nElems; j++) // buscar elementos
68 if(arr.getElem(j) == 55)
69         break;
70 for(int k=j; k<nElems; k++) // mover higher ones down
71                 arr.setElem(k, arr.getElem(k+1) );
72         nElems--; // decrementar el size
73
74 //--------------------------------------
75
76 for(j=0; j<nElems; j++) // display items
77         System.out.print( arr.getElem(j) + " ");
78 System.out.println("");
79 } // fin de main()
80 } // fin de la clase LowArrayApp
```

[5, pág,39]

Abstracción El proceso de separar el cómo del qué, cómo se realiza una operación dentro de una clase, a diferencia de lo que es visible para el usuario de la clase, se llama abstracción. Abstracción es un aspecto importante de la ingeniería de software. Al abstraer la funcionalidad de la clase hacemos que sea más fácil diseñar un programa, porque no necesitamos pensar en detalles de implementación en una etapa demasiado temprana del proceso de diseño.

[5, pág,45]

Interface del libro Algoritmos Java proporciona soporte de lenguaje para definir relaciones entre objetos, conocido como **herencia**. Estos mecanismos son ampliamente utilizados por el desarrolladores de software, por lo que los estudiará en detalle si toma un curso de ingeniería de software En g. El primer mecanismo de herencia que consideramos se conoce como subtipo, que nos permite especificar una relación entre clases que de otro modo no estarían relacionadas al especificar en una interfaz un conjunto de métodos comunes que debe contener cada clase de implementación. **Un interfaz no es más que una lista de métodos de instancia**. Por ejemplo, en lugar de usar nuestra API informal, podríamos haber articulado una interfaz para Fecha:

Listing 56: Ejemplo de Interface en java.

```
1 public interface Datable {
2         int anio();
3         int mes();
4         int dia();
5 }
```

y luego se refiere a la interfaz en nuestro código de implementación como a continuación se muestra:

Listing 57: Uso de la Interface en java.

```
1 public class Date implements Datable {
2         // implementacion de codigo (paraecido al de antes)
3 }
```

para que el compilador de Java verifique que coincida con la interfaz. Agregue el código **implements Datable** a cualquier clase que implemente mes (), día () y año (). Esto ofrece una garantía a cualquier cliente de que un objeto de esa clase puede invocar esos métodos. Esta disposición se conoce como herencia de interfaz: una clase de implementación hereda el interfaz. La herencia de interfaz nos permite escribir programas de cliente que pueden manipular objetos de cualquier tipo que implemente la interfaz (incluso un tipo para ser creado en el futuro), al invocar métodos en la interfaz.

java.lang.Comparable compareTo () 2.1

Podríamos tener herencia de interfaz utilizada en lugar de

java.util.Comparator comparar() 2.5

nuestras API más informales, pero elegimos no hacerlo para evitar la dependencia de lenguaje específico de alto nivel

java.lang.Iterable iterador () 1.3

nismos que no son críticos para el iteración

hasNext ()

comprensión de algoritmos y

java.util.Iterator próximo() 1.3

para evitar el equipaje extra de inter eliminar() archivos de cara. Pero hay algunas situaciones. Interfaces Java utilizadas en este libro donde las convenciones de Java hacen Merece la pena aprovechar las interfaces: las usamos para comparar y para iteración, como se detalla en la tabla al final de la página anterior, y considerará ellos con más detalle cuando cubrimos esos conceptos.

[8, pág, 100]

Más sobre interface Para distinguir entre una interface y una clase, se muestra en el siguiente ejemplo la interfaz comestible (edible):

Listing 58: Ejemplo de Interface en java.

```
1  public interface Comestible {
2          /** Describe como comer */
3          public abstract String howToEat();
4  }
```

Listing 59: Ejemplo de Interface heredada en java.

```
1  public class TestComestible {
2          public static void main(String[] args) {
3                  Object[] objects = {new Tiger(), new
4                  Chicken(),
5                  new Apple()};
6
7            for (int i = 0; i < objects.length; i++)
8                  if (objects[i] instanceof Edible)
9                          System.out.println(((Edible)
10                         objects[i]).howToEat());
11         }
12 }
13
14 class Animal {
15         // Campos de datos, constructores, y metodos
16         omitidos aqui
17 }
18 class Chicken extends Animal implements Edible {
19         public String howToEat() {
20                 return "Chicken: Fry it";
21         }
22 }
23 class Tiger extends Animal {
24 }
25 abstract class Fruit implements Comestible {
26         // Campos de datos, constructores, y metodos
27          omitidos aqui
28 }
29 class Apple extends Fruit {
30         public String howToEat() {
31         return "Apple: Make apple cider";
32         }
33 }
34 class Orange extends Fruit {
35         public String howToEat() {
36                 return "Orange: Make orange juice";
37         }
38 }
```

El método compareTo determina el orden de este objeto con el objeto especificado .ºz regresa un entero negativo, cero, o un entero positivo si este objeto es menor que, igual a, o más grande que .º".

Listing 60: Ejemplo de Interface en java.

```
1  // Interface for comparing objects, defined in java.lang
2  package java.lang;
3
4  public interface Comparable {
5          public int compareTo(Object o);
6  }
```

Muchas clases en las librerías de java (es decir String y Date) implementan **Comparable** para definir un orden natural de los objetos. Si examinamos el código fuente de estas clases, usted verá la palabra clave **implements** usado en las clases:

Listing 61: Ejemplo de uso de Interface en java.

```
1  public class String extends Object
2          implements Comparable {
3                  // class body omitted
4  }
```

Listing 62: Ejemplo de uso de Interface en java.

```
1  public class Date extends Object
2          implements Comparable {
3                  // class body omitted
4  }
```

Listing 63: Ejemplo de uso de Interface en java.

```
1  public class ComparableRectangle extends Rectangle
2  implements Comparable {
3          /** Construct a ComparableRectangle with
4          specified properties */
5          public ComparableRectangle(double width,
6          double height) {
7                          super(width, height);
8          }
9
10         /** Implement the compareTo method defined
11         in Comparable */
12         public int compareTo(Object o) {
13                 if (getArea() > ((ComparableRectangle)o)
14                 .getArea())
15                         return 1;
16                 else if (getArea() < ((
17                 ComparableRectangle)o).getArea())
18                         return -1;
```

```
19                  else
20                          return 0;
21          }
22  }
```

Listing 64: Ejemplo de uso de Interface en java.

```
1  public class TestGeometricObject {
2     /** Main method */
3     public static void main(String[] args) {
4
5     // Create two geometric objects
6     GeometricObject geoObject1 = new Circle(5);
7     GeometricObject geoObject2 = new Rectangle(5, 3);
8
9     System.out.println("The two objects have the
10 ___same area? " +
11    equalArea(geoObject1, geoObject2) );
12
13    // Display circle
14    displayGeometricObject(geoObject1);
15
16    // Display rectangle
17    displayGeometricObject(geoObject2);
18 }
19
20    /** A method for comparing the areas of two geometric
21     objects */
22    public static boolean equalArea(GeometricObject
23     object1,
24    GeometricObject object2) {
25       return object1.getArea() == object2.getArea();
26    }
27
28    /** A method for displaying a geometric object */
29    public static void displayGeometricObject(
30    GeometricObject object) {
31       System.out.println();
32       System.out.println("The area is " + object.
33       getArea());
34       System.out.println("The perimeter is " +
35       object.getPerimeter());
36    }
37 }
```

Listing 65: Ejemplo de uso de Interface en java.

```
1  import java.util.*;
2
3  public class TestCalendar {
```

```
4     public static void main(String[] args) {
5     // Construct a Gregorian calendar for the
6      current date and time
7     Calendar calendar = new GregorianCalendar();
8     System.out.println("Current time is " + new Date());
9     System.out.println("YEAR:\t" + calendar.get(
10    Calendar.YEAR));
11    System.out.println("MONTH:\t" + calendar.get(
12    Calendar.MONTH));
13    System.out.println("DATE:\t" + calendar.get(
14    Calendar.DATE));
15    System.out.println("HOUR:\t" + calendar.get(
16    Calendar.HOUR));
17    System.out.println("HOUR_OF_DAY:\t" +
18    calendar.get(Calendar.HOUR_OF_DAY));
19    System.out.println("MINUTE:\t" + calendar.get(
20    Calendar.MINUTE));
21    System.out.println("SECOND:\t" + calendar.get(
22    Calendar.SECOND));
23    System.out.println("DAY_OF_WEEK:\t" +
24    calendar.get(Calendar.DAY_OF_WEEK));
25    System.out.println("DAY_OF_MONTH:\t" +
26    calendar.get(Calendar.DAY_OF_MONTH));
27    System.out.println("DAY_OF_YEAR: " +
28    calendar.get(Calendar.DAY_OF_YEAR));
29    System.out.println("WEEK_OF_MONTH: " +
30    calendar.get(Calendar.WEEK_OF_MONTH));
31    System.out.println("WEEK_OF_YEAR: " +
32    calendar.get(Calendar.WEEK_OF_YEAR));
33    System.out.println("AM_PM: " + calendar.get(
34    Calendar.AM_PM));
35
36    // Construct a calendar for September 11, 2001
37    Calendar calendar1 = new GregorianCalendar(2001, 8,
38     11);
39    System.out.println("September 11, 2001 is a " +
40    dayNameOfWeek(calendar1.get(Calendar.DAY_OF_WEEK)));
41    }
42
43    public static String dayNameOfWeek(int dayOfWeek) {
44       switch (dayOfWeek) {
45          case 1: return "Sunday";
46          case 2: return "Monday";
47          case 3: return "Tuesday";
48          case 4: return "Wednesday";
49          case 5: return "Thursday";
50          case 6: return "Friday";
51          case 7: return "Saturday";
52          default: return null;
53       }
```

```
54     }
55  }
```

Listing 66: Ejemplo de uso de Interface en java.

```
1  import javax.swing.*;
2  import java.awt.event.*;
3  public class HandleEvent extends JFrame {
4          public HandleEvent() {
5                  // Create two buttons
6                  JButton jbtOK = new JButton("OK");
7                  JButton jbtCancel = new JButton(
8                  "Cancel");
9                  // Create a panel to hold buttons
10                 JPanel panel = new JPanel();
11                 panel.add(jbtOK);
12                 panel.add(jbtCancel);
13                 add(panel); // Add panel to the frame
14                 // Register listeners
15                 OKListenerClass listener1 = new
16                 OKListenerClass();
17                 CancelListenerClass listener2 = new
18                 CancelListenerClass();
19                 jbtOK.addActionListener(listener1);
20                 jbtCancel.addActionListener(listener2);
21         }
22         public static void main(String[] args) {
23                 JFrame frame = new HandleEvent();
24                 frame.setTitle("Handle Event");
25                 frame.setSize(200, 150);
26                 frame.setLocation(200, 100);
27                 frame.setDefaultCloseOperation(JFrame.
28                 EXIT_ON_CLOSE);
29                 frame.setVisible(true);
30         }
31 }
32 class OKListenerClass implements ActionListener {
33         public void actionPerformed(ActionEvent e) {
34                 System.out.println("OK button clicked");
35         }
36 }
37 class CancelListenerClass implements ActionListener {
38         public void actionPerformed(ActionEvent e) {
39                 System.out.println(
40                 "Cancel button clicked");
41         }
42 }
```

[6, pág, 468]

Del manual de java sobre interface Usando la palabra clave **interface**, puede abstraer completamente la interfaz de una clase desde su implementación. Es decir, utilizando **interface**, puede especificar qué debe hacer una clase, pero no cómo lo hace. Las **interface**s son sintácticamente similares a las clases, pero carecen de variables de instancia y sus métodos son declarados sin ningún cuerpo. En la práctica, esto significa que se pueden definir **interface**s para no hacer suposiciones sobre cómo se implementan. Una vez que se define, cualquier número de Las clases se puede implementar una interfaz. Además, una clase puede implementar cualquier número de interfaces. Para implementar una interfaz, una clase debe crear el conjunto completo de métodos definidos por la interfaz. Sin embargo, cada clase es libre de determinar los detalles de su propia implementación. Al proporcionar la palabra clave **interface**, Java le permite utilizar completamente "una **interface**, con múltiples métodos" aspecto del polimorfismo.

Las interfaces están diseñadas para admitir la resolución dinámica de métodos en tiempo de ejecución. Normalmente, para que un método se llame de una clase a otra, ambas clases deben estar presentes en tiempo de compilación para que el compilador de Java pueda verificar que las firmas del método sean compatibles. Este requisito por sí solo hace una clasificación estática y no extensible ambiente. Inevitablemente en un sistema como este, la funcionalidad se eleva cada vez más en la jerarquía de clases para que los mecanismos estén disponibles para más y más subclases. Las interfaces están diseñadas para evitar este problema. Desconectan la definición de un método o conjunto de métodos de la jerarquía de herencia. Dado que las interfaces están en una jerarquía diferente de las clases, es posible que las clases que no están relacionadas en términos de la jerarquía de clases se implementen con la misma interfaz. Aquí es donde se realiza el verdadero poder de las interfaces.

Definición de interface Una interfaz se define como una clase. Esta es la forma general de una interfaz:

Listing 67: Forma general de una interfaz en java.

```
1 access interface name {
2         return-type method-name1(lista de parametros);
3         return-type method-name2(lista de parametros);
4         type final-varname1 = value;
5         type final-varname2 = value;
6         // ...
7         return-type method-nameN(lista de parametros);
8         type final-varnameN = value;
9 }
```

Cuando no se incluye un especificador de acceso, los resultados de acceso predeterminados, y la interfaz es solo disponible para otros miembros del paquete en el que se declara. Cuando se declara como público, la interfaz puede ser utilizada por cualquier otro código. En este caso, la interfaz debe ser la solo la interfaz pública declarada en el archivo, y el archivo debe tener el mismo nombre que la interfaz. nombre es el nombre de la interfaz y puede ser cualquier identificador válido. Tenga en cuenta que los métodos que son declarados no tienen cuerpos. Terminan con un punto y coma después de la lista de parámetros. Son,

esencialmente, métodos abstractos; no puede haber una implementación predeterminada de ningún método especificado dentro de una interfaz. Cada clase que incluye una interfaz debe implementar todos los métodos. Las variables se pueden declarar dentro de las declaraciones de interfaz. Son implícitamente finales y estático, lo que significa que la clase implementadora no puede cambiarlos. También deben ser inicializado Todos los métodos y variables son implícitamente públicos. Aquí hay un ejemplo de una definición de interfaz. Declara una interfaz simple que contiene Un método llamado callback () que toma un único parámetro entero.

Listing 68: Ejemplo de uso de Interface en java.

```
1 interface Callback {
2         void callback(int param);
3 }
```

Implementando interfaces Una vez que se ha definido una interfaz, una o más clases pueden implementar esa interfaz. A implementar una interfaz, incluir la cláusula implements en una definición de clase y luego crear Los métodos definidos por la interfaz. La forma general de una clase que incluye los implementos. La cláusula se ve así:

Listing 69: Ejemplo de uso de Interface en java.

```
1 class classname [extends superclass] [implements
2 interface [,interface...]] {
3         // cuerpo de la class
4 }
```

Si una clase implementa más de una interfaz, las interfaces se separan con una coma. Si una clase implementa dos interfaces que declaran el mismo método, entonces el mismo método ser utilizado por clientes de cualquiera de las interfaces. Los métodos que implementan una interfaz deben ser declarado público, además, la firma de tipo del método de implementación debe coincidir exactamente la firma de tipo especificada en la definición de interfaz. Aquí hay una pequeña clase de ejemplo que implementa la interfaz de devolución de llamada mostrada anteriormente.

Listing 70: Ejemplo de uso de Interface en java.

```
1 class Client implements Callback {
2         // Implemento de la interface de retorno
3         public void callback(int p) {
4                 System.out.println("llamada␣retornada
5 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣con␣" + p);
6         }
7 }
```

Observe que callback() se declara utilizando el especificador de acceso **public**. Es permisible y común para las clases que implementan interfaces para definir miembros adicionales propios. Por ejemplo, la siguiente versión de cliente implementa callback () y agrega el método nonIfaceMeth ():

Listing 71: Ejemplo de uso de Interface en java.

```
1  class Client implements Callback {
2          // Implemento de la interface de retorno
3          public void callback(int p) {
4                  System.out.println("llamada␣retornada
5  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣con␣" + p);
6          }
7          void nonIfaceMeth() {
8                  System.out.println("Classes␣que
9  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣implementan␣interfaces␣" +
10                 "puede␣tambien␣definir␣otros␣miembros,
11 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣otra␣vez.");
12         }
13 }
```

[7, pág, 194]

La clase Object del libro de Dathan Java tiene una clase especial llamada **Object** de la cual cada clase hereda. En otras palabras, **Object** es una superclase de todas las clases en Java y está en la raíz de la jerarquía de clases. De nuestro conocimiento de asignaciones polimórficas, podemos ver que una variable de tipo **Object** puede almacenar la referencia a un objeto de cualquier otro tipo. El siguiente código es, por lo tanto, legal.

Listing 72: Ejemplo del uso de Object en java.

```
1 Object anyObject;
2 anyObject = new Student();
3 anyObject = new Integer(4);
4 anyObject = "Some␣string";
```

En lo anterior, la variable **anyObject** primero almacena un objeto Estudiante, luego un objeto Entero, y finalmente un objeto String.

La clase Object del libro de Dean [pás. 510] La clase Object es el ancestro de todas las otras clases. Es el ancestro primordial, la raíz de la jerarquía de la herencia. Cualquier clase que extienda explícitamente una superclase utiliza extends en su definición. Cuando cualquiera crea una nueva clase que no extiende explícitamente alguna otra clase, el compilador automáticamente hace que extienda la clase Object. Por lo tanto, todas las clases eventualmente descienden de la clase Object. La clase Object no tiene muchos métodos, pero los que tiene son significativos, porque siempre son heredados por todas las otras clases. En las siguientes dos secciones, verá los dos métodos más importantes de la clase Object, equals y toString. Como cualquier clase que escriba incluye automáticamente estos dos métodos, debe tener en cuenta lo que sucede cuando se llaman estos métodos. Sin embargo, antes de sumergirnos en los detalles de estos dos métodos, queremos informarle sobre un programa Java proceso que es muy similar a la promoción de tipo numérico que estudiaste en el Capítulo 3 y el Capítulo 11. Ahí vimos al hacer una tarea o copiar un argumento en un parámetro, el Java Virtual Machine (JVM) promueve automáticamente un tipo numérico, siempre que el cambio se ajuste a un determinado jerarquía

numérica, por ejemplo, cuando se asigna un valor int a una variable doble, la JVM autopromueve matemáticamente el valor int a un valor doble. Una promoción automática análoga también ocurre con otros tipos. Cuando una tarea o argumento la operación de paso implica diferentes tipos de referencia, la JVM promueve automáticamente la referencia de origen escriba el tipo de referencia de destino si el tipo de referencia de destino está por encima del tipo de referencia de origen en la jerarquía de la herencia. En particular, dado que la clase Object es un antepasado de cualquier otra clase, cuando la necesidad surge, Java automáticamente promueve cualquier tipo de clase al tipo de objeto. La siguiente sección describe una situación que estimula este tipo de promoción.

El método equals de la clase Object del libro de Dean [pás. 510] El método igual de la clase Object, que todas las demás clases heredan automáticamente, tiene esta interfaz pública:

Listing 73: Ejemplo del uso de Object en java.

```
1 public boolean equals(Object obj);
```

Como todas las clases heredan automáticamente este método, a menos que un método definido de manera similar tenga prioridad, cualquier objeto, objectA, puede invocar este método para compararse con cualquier otro objeto, objectB, con un método llamado así:

Listing 74: Ejemplo del uso de Object en java.

```
1 objectA.equals(object B);
```

Esta llamada al método devuelve un valor booleano de verdadero o falso. Tenga en cuenta que no especificamos el tipo de objeto A u objeto B. En general, pueden ser instancias de cualquier clase, y no necesitan ser objetos de la misma clase. La única restricción es que objectA debe ser una referencia no nula. Por ejemplo, si existen clases de perros y gatos, este código funciona correctamente:

Listing 75: Ejemplo del uso de Object en java.

```
1 Cat cat = new Cat();
2 Dog dog = new Dog();
3 System.out.println(cat.equals(dog));
```

El método igual que se llama aquí es el método igual que la clase Cat automáticamente hereda de la clase Object. El parámetro en este método heredado es de tipo Object, como se especifica en la interfaz pública del método anterior. Pero el argumento del perro que pasamos a este método no es de tipo Object. Eso es de tipo perro. ¿Entonces que esta pasando? Cuando pasamos la referencia del perro al método igual heredado, el tipo de referencia pasa automáticamente de tipo Perro a tipo Objeto. Entonces lo heredado es igual El método realiza una prueba interna para ver si el pase del perro es el mismo que el del gato que llama. Por supuesto que es no, entonces la salida es falsa, como puedes ver.

Herencia en una interfaz del libro Dathan página 62 Un tipo especializado de herencia es aquel en el que una clase hereda de una interfaz. Recolección en el Capítulo 2 habíamos definido una interfaz como una colección de métodos

que pueden ser implementado por una clase. Se ha comparado una interfaz con un contrato firmado por la clase. implementando la interfaz. En el contexto de este capítulo, debe señalarse que la implementación de la interfaz también se puede ver como una forma de herencia, donde la implementación La clase menting hereda un conjunto abstracto de propiedades de la interfaz. Java reconoce una interfaz como un tipo (al igual que otros lenguajes orientados a objetos), lo que significa que los objetos que pertenecen a clases que implementan una interfaz dada también pertenecen al tipo representado por la interfaz. Del mismo modo, podemos declarar un identificador como perteneciente al tipo de interfaz y luego podemos usarla para acceder a los objetos de cualquier clase que implementa la interfaz

Listing 76: Ejemplo del uso de herencia en Interface en java.

```
1  public interface I {
2  // details of I
3  }
4  public class A implements I {
5  //code for A
6  }
7  public class B implements I {
8  //code for B
9  }
10 I i1 = new A(); // i1 holds an A
11 I i2 = new B(); // i2 holds a B
```

En la notación UML, este tipo de relación entre la interfaz y la implementación de la clase, se denomina realización y está representada por una línea de puntos con una gran punta de flecha abierta que apunta a la interfaz como se muestra en la Figura 3.8.

Polimorfismo y enlace dinámico del libre de Dathan Considere una aplicación universitaria que contiene, entre otras, tres clases que forman un jerarquía como se muestra en la Figura 3.9. Un estudiante puede ser un estudiante universitario o un estudiante graduado. Al igual que en la vida real, donde pensaríamos en un estudiante universitario o un estudiante graduado como estudiante, en el paradigma orientado a objetos también, consideramos un El objeto de estudiante de grado o un objeto de estudiante de grado debe ser del tipo Estudiante. Por lo tanto, podemos escribir

Listing 77: Ejemplo del uso de polimorfismo en java.

```
1  Student student1 = new UndergraduateStudent();
2  Student student2 = new GraduateStudent();
```

Esta es una idea poderosa. Ahora podemos escribir métodos que acepten un Estudiante y aprueben un estudiante de pregrado o un estudiante de posgrado a él como se muestra a continuación.

Listing 78: Método para almacenar un estudiante en java.

```
1  public void storeStudent(Student student) {
2          // code to store the Student object
```

```
3 }
```

Libro orientado a objetos 52-54 Relaciones entre clases

En el capítulo anterior estudiamos clases y objetos como los dos bloques de construcción de sistemas orientados. La estructura de un sistema de software se define por la forma en que estos los bloques se relacionan entre sí y el comportamiento del sistema está definido por el En el capítulo anterior, estudiamos clases y objetos como los dos bloques de construcción de objetos. manera en que los objetos interactúan entre sí. Por lo tanto, para construir un sistema de software, necesitamos mecanismos que creen conexiones entre estos edificios bloques En este capítulo presentamos los tipos básicos de relaciones entre clases (y objetos) que hacen las conexiones.

El tipo de relaciones más simple y general es la asociación, que simplemente indica indica que los objetos de las dos clases están relacionados de alguna manera no jerárquica. Allí casi no hay otras restricciones sobre cómo se puede formar una asociación, aunque lo haremos vea a lo largo de este texto las buenas prácticas de diseño que se deben seguir al crear asociaciones.

Cuando dos o más clases tienen una relación jerárquica basada en la generalización, se conoce como herencia. Las clases conectadas por herencia comparten algunos puntos en común y por lo tanto, este tipo de relación es más restrictiva que la asociación.

El tercer tipo de relación que vemos es la genérica. Esto es más restrictivo que la herencia. debido a que las únicas variaciones permitidas en las clases relacionadas son aquellas que se puede capturar mediante parametrización de tipos, es decir, proporcionando parámetros de diferentes tipos al crear una instancia de la entidad genérica.

En el resto de este capítulo elaboramos cada uno de estos, discutiendo los principios básicos. y examinar situaciones en las que se pueden aplicar. Dado que estos mecanismos son básicos para OOAD, todos serán revisados en capítulos posteriores cuando se trate de ejemplos reales de más sistemas complejos.

Asociación

Una asociación se define formalmente como una relación entre dos o más clases que describe un grupo de enlaces con estructura común y semántica. Una asociación implica que un objeto de una clase está haciendo uso de un objeto de otra clase y se indica simplemente por un sólido línea que conecta los dos íconos de clase. En el capítulo anterior definimos una clase Estudiante que realiza un seguimiento de la información sobre los cursos para los que el estudiante se ha registrado. Esta La información se representa como se muestra en la Figura 3.1. En nuestro ejemplo, los objetos de estudiante pueden hacer uso de los objetos del curso cuando se generan transcripciones, cuando se calcula la matrícula o cuando se descarta un curso. El enlace al curso proporciona al estudiante objeto con el Información necesaria.

Una asociación no implica que siempre haya un enlace entre todos los objetos de una clase y todos los objetos del otro. Como cabría esperar, en nuestro ejemplo, se forma un enlace antes de entre un objeto Estudiante y un objeto Curso solo cuando la operación que los vincula es completado, es decir, el estudiante representado por el objeto Estudiante se registra para ese particular curso. Sin embargo, una asociación implica que hay una conexión persistente e identificable ción entre dos clases. Si la clase A está asociada con la clase B,

significa que dado un objeto de clase A, siempre puede encontrar un objeto de clase B, o puede encontrar que ningún objeto B tiene sido asignado a la asociación todavía. Pero en cualquier caso siempre hay un camino identificable de A a B. Las asociaciones también representan relaciones conceptuales entre clases como relaciones físicas entre los objetos de estas clases. En términos de implementación, lo que implica lo anterior es que la clase A debe proporcionar un mecanismo usando las construcciones del lenguaje de programación elegido para formar un enlace. Esto podría tomar varias formas, por ejemplo,

• La clase A almacena una clave (o varias claves) que identifica de forma exclusiva un objeto de la clase B. • La clase A almacena una referencia (s) a objeto (s) de la clase B. • La clase A tiene una referencia a un objeto de la clase C, que, a su vez, está asociado con un instancia única de clase B.

Los dos primeros de estos crean una asociación directa, mientras que el tercero es una asociación indirecta. El mecanismo elegido puede depender de los requisitos que el sistema debe cumplir, satisfacer (por ejemplo, los tipos de consultas que deben responderse) y también sobre cómo el resto del sistema está diseñado. En nuestro ejemplo, cuando un estudiante se inscribe en un curso, él / ella en realidad se inscribe en una sección específica del curso. El mecanismo para hacer esta conexión puede ser simplemente que el objeto Estudiante almacene una referencia al objeto Sección. La sección está asociada con un curso único, completando la imagen (ver Figura 3.2).

Se supone que una asociación es bidireccional a menos que coloquemos una flecha direccional en la línea de conexión para indicar lo contrario. La asociación generalmente tiene un nombre descriptivo. El nombre a menudo implica una dirección, pero en la mayoría de los casos esto puede invertirse. Nuestra figura dice que el estudiante se matricula en una sección, que pertenece a un curso, pero esto se podría indicar como El curso tiene secciones que inscriben estudiantes. El diagrama generalmente se dibuja para leer el enlace o asociación de izquierda a derecha o de arriba a abajo. Las entidades al final de la asociación generalmente tienen roles asignados, que pueden tener nombres Podríamos tener una asociación llamada .$^{\text{em}}$plea"que conecta una clase que representa un Business a una clase que representa a una persona empleada por la empresa. Here Business desempeña el papel del empleador y la persona tiene el papel de empleado.

El elemento try pág 205 manual de java Una excepción de Java es un objeto que describe una condición excepcional (es decir, error) que ocurrió en una pieza de código. Cuando surge una condición excepcional, un objeto que representa esa excepción se crea y se lanza en el método que causó el error. Ese método puede elige manejar la excepción en sí misma o pasarla. De cualquier manera, en algún momento, la excepción es atrapado y procesado. El sistema de tiempo de ejecución de Java puede generar excepciones o puede ser generado manualmente por su código. Las excepciones lanzadas por Java se relacionan con fundamental errores que violan las reglas del lenguaje Java o las restricciones de la ejecución de Java ambiente. Las excepciones generadas manualmente se usan generalmente para informar alguna condición de error a la persona que llama de un método. El manejo de excepciones de Java se gestiona a través de cinco palabras clave: try, catch, throw, throws y finalmente. Brevemente, así es como funcionan. Declaraciones de programa que desea supervisar las excepciones

están contenidas dentro de un bloque try. Si ocurre una excepción dentro del bloque try, es arrojado Su código puede detectar esta excepción (usando catch) y manejarla de alguna manera racional. El sistema de tiempo de ejecución de Java genera automáticamente excepciones generadas por el sistema. A lanzar manualmente una excepción, use la palabra clave throw. Cualquier excepción que se descarte un método debe especificarse como tal mediante una cláusula throws. Cualquier código que absolutamente debe ser ejecutado después de que se complete un bloque de prueba se coloca en un bloque finalmente. Esta es la forma general de un bloque de manejo de excepciones:

try { // block de código para monitorear los errores }

Libro pensando en java página 408 Si uno está dentro de un método y lanza una excepción (o lo hace otro método al que se invoque), ese método acabará en el momento en que haga el lanzamiento. Si no se desea que una excepción implique abandonar un método, se puede establecer un bloque especial dentro de ese método para que capture la excepción. A este bloque se le denomina el bloque try puesto que en él se ïntentan"varias llamadas a métodos. El bloque try es un ámbito ordinario, precedido de la palabra clave try:

try { / / Código que podría generar excepciones }

Si se estuviera comprobando la existencia de errores minuciosamente en un lenguaje de programación que no soporte manejo de excepciones, habría que rodear cada llamada a método con código de prueba de invocación y errores, incluso cuando el mismo método fuese invocado varias veces. Esto significa que el código es mucho más fácil de escribir y leer debido a que no se confunde el objetivo del código con la comprobación de errores.

Excepciones en constructores pág 430 pensando en java Cuando se escribe código con excepciones, es particularmente importante que siempre se pregunte: "Si se da una excepción, ¿será limpiada adecuadamente?"La mayoría de veces es bastante seguro, pero en los constructores hay un problema. El constructor pone el objeto en un estado de partida seguro, pero podría llevar a cabo otra operación -como abrir un fichero- que no se limpia hasta que el usuario haya acabado con el objeto y llame a un método de limpieza especial. Si se lanza una excepción desde entro de un constructor, puede que estos comportamientos relativos a la limpieza no se den correctamente. Esto significa que hay que ser especialmente cuidadoso al escribir constructores.

Dado que se acaba de aprender lo que ocurre con finally, se podría pensar que es la solución correcta. Pero no es tan simple, puesto que finally ejecuta siempre el código de limpieza, incluso en las situaciones en las que no se desea que se ejecute este código de limpieza hasta que acabe el método de limpieza. Por consiguiente, si se lleva a cabo una limpieza en finally, hay que establecer algún tipo de indicador cuando el constructor finaliza normalmente, de forma que si el indicador está activado no se ejecute nada en finally. Dado que esto no es especialmente elegante (se está asociando el código de un sitio a otro), es mejor si se intenta evitar llevar a cabo este tipo de limpieza en el método finally, a menos que uno se vea forzado a ello.

En el ejemplo siguiente, se crea una clase llamada ArchivoEntrada que abre un archivo y permite leer una línea (convertida a String) de una vez. Usa las clases FileReader y BufferedReader de la biblioteca estándar de E/S de Java

que se verá en el Capítulo 11, pero que son lo suficientemente simples como para no tener ningún problema en tener su uso básico:

Orden de inicialización pág 153 pensando en java Dentro de una clase, el orden de inicialización lo determina el orden en que se definen las variables dentro de la clase. Las definiciones de variables pueden estar dispersas a través y dentro de las definiciones de métodos, pero las variables se inicializan antes de invocar a ningún método -incluido el constructor. Por ejemplo:

Bibliografía

[1] Balagurusamy, E. *Object oriented programming with C++*, fourth edition ed. McGraw-Hill Publishing Company Limited, New Delhi, 2008.

[2] Dathan, B., and Ramnath, S. *Object-Oriented Analysis and Design.* Springer, 2008.

[3] Dean, J., and Dean, R. *Introduction to programming with java: a problem solving approach.* McGraw Hill Book Company.

[4] Eckel, B. *Thinking in C++*, second ed. Prentice Hall, 2000.

[5] Lafore, R. *Data Structures & Algorithms in Java.* 1998.

[6] Liang, Y. D. *Introduction to java programming*, eighth edition ed. Prentice Hall, 2011.

[7] Schildt, H. *Java : The Complete Reference*, seventh edition ed., 2007.

[8] Sedgewick, R., and Wayne, K. *Algorithms*, fourth edition ed. Addison Wesley, 2011.

[9] Torczon, L., and Cooper, K. D. *Engineering a Compiler*, second ed., vol. 13. Elsevier, 2012.