Teoría de Objetos en Java

Deitel P, **Deitel H**. *Java Como Programar* 10 ed

Introducción a la tecnología de los objetos

Ahora que la demanda de software nuevo y más poderoso va en aumento, crear software en forma rápida, correcta y económica sigue siendo un objetivo difícil de alcanzar. Los *objetos*, o dicho en forma más precisa, las clasess de las que provienen los objetos, son en esencia componentes de software *reutilizables*. Existen objetos de fecha, objetos de hora, objetos de audio, objetos de video, objetos de automóviles, objetos de personas, etc. Casi cualquier *sustantivo* se puede representar de manera razonable como un objeto de software en términos de sus *atributos* (como el nombre, color y tamaño) y *comportamientos* (por ejemplo, calcular, moverse y comunicarse). Los grupos de desarrollado de software pueden usar una metodología de diseño e implementación orientada a objetos y modular para ser mucho más productivos de lo que era posible con las técnicas populares anteriores, como la “programación estructurada”. Por lo general los programas orientados a objetos son más fáciles de comprender, corregir y modificar.

El automóvil como un objeto

Para ayudarle a comprender los objetos y su contenido, empecemos con una analogía simple. Suponga que *desea conducir un auto y hacer que vaya más rápido al presionar el pedal del acelerador*. ¿Qué debe ocurrir para que usted pueda hacer esto? Bueno, antes de que pueda conducir un auto, alguien tiene que *diseñarlo*. Por lo general, un auto empieza en forma de dibujos de ingeniería, similares a los *planos de construcción* que describen el diseño de una casa. Estos dibujos de ingeniería incluyen el diseño del pedal del acelerador. El pedal *oculta* al conductor los complejos mecanismos que se encargan de que el auto aumente su velocidad, de igual forma que el pedal del freno “oculta” los mecanismos que disminuyen la velocidad del auto y el volante “oculta” los mecanismos que hacen que el auto de vuelta. Esto permite que las personas con poco o ningún conocimiento sobre cómo funcionan los motores, los frenos y los mecanismos de la dirección puedan conducir un auto con facilidad.

Así como no es posible cocinar en la cocina que está en un plano de construcción, tampoco es posible conducir los dibujos de ingeniería de un auto. Antes de poder conducir un auto, éste debe construirsee a partir de los dibujos de ingeniería que lo describen. Un auto completo tendrá un pedal acelerador *verdadero* para hacer que aumente su velocidad, pero aun así no es suficiente; el auto no acelerará por su propia cuenta (¡esperemos que así sea!), así que el conductor debe *presionar* el pedal para acelerar el auto.

Métodos y clases

Ahora vamos a utilizar nuestro ejemplo del auto para presentar algunos conceptos clave de la programación orientada a objetos. Para realizar una tarea en una aplicación se requiere un **método**. Ese método aloja las instrucciones del programa que se encargan de realizar sus tareas. El método oculta al usuario estas instrucciones, de la misma forma que el pedal del acelerador de un auto oculta al conductor los mecanismos para hacer que el auto vaya más rápido. En Java, creamos una unidad de programa llamada **clase** para alojar el conjunto de métodos que realizan las tareas de esa clase. Por ejemplo, una clase que representa a una cuenta bancaria podría contener un método para *depositar* dinero en una cuenta, otro para

retirar dinero de una cuenta y un tercero para *solicitar* el saldo actual de la cuenta. Una clase es similar en concepto a los dibujos de ingeniería de un auto, que contienen el diseño de un pedal acelerador, volante de dirección, etcétera.

Instanciación

Así como alguien tiene que construir un *auto* a partir de sus dibujos de ingeniería para que alguien más pueda conducirlo después, también es necesario *crear un objeto* de una clase para que un programa pueda realizar las tareas definidas por los métodos de esa clase. Al proceso de hacer esto se le denomina instanciación. Por lo tanto, un objeto viene siendo una **instancia** de su clase.

Reutilización

Así como los dibujos de ingeniería de un auto se pueden *reutilizar* muchas veces para construir muchos autos, también es posible *reutilizar* una clase muchas veces para crear muchos objetos. Al reutilizar las clases existentes para crear nuevas clases y programas, ahorramos tiempo y esfuerzo. La reutilización también nos ayuda a crear sistemas más confiables y efectivos, ya que con frecuencia las clases y los componentes existentes pasan por un extenso proceso de *prueba*, *depuración* y optimización del *desempeño*. De la misma manera en que la noción de *piezas intercambiables* fue crucial para la Revolución Industrial, las clases reutilizables son cruciales para la revolución del software incitada por la tecnología de objetos.

Mensajes y llamadas a métodos

Cuando usted conduce un auto, al presionar el pedal del acelerador envía un *mensaje* al auto para que realice una tarea: aumentar la velocidad. De manera similar, es posible *enviar mensajes a un objeto*. Cada mensaje se implementa como **llamada a método**, para indicar a un método del objeto que realice su tarea. Por ejemplo, un programa podría llamar al método *depositar* de un objeto cuenta de banco para aumentar el saldo de esa cuenta.

Atributos y variables de instancia

Además de tener capacidades para realizar tareas, un auto también tiene *atributos*: color, número de puertas, capacidad de gasolina en el tanque, velocidad actual y registro del total de kilómetros recorridos (es decir, la lectura de su odómetro). Al igual que sus capacidades, los atributos del auto se representan como parte de su diseño en sus diagramas de ingeniería (que, por ejemplo, incluyen un velocímetro y un indicador de combustible). Al conducir un auto real, estos atributos se llevan junto con el auto. Cada auto mantiene sus *propios* atributos. Por ejemplo, cada uno sabe cuánta gasolina hay en su tanque, pero *no* cuánta hay en los tanques de *otros* autos.

De manera similar, un objeto tiene atributos que lleva consigo a medida que se utiliza en un programa. Estos atributos se especifican como parte del objeto de esa clase. Por ejemplo, un objeto *cuenta bancaria* tiene un *atributo saldo* que representa la cantidad de dinero en la cuenta. Cada objeto cuenta bancaria conoce el saldo de la cuenta que representa, pero *no* los saldos de las *otras* cuentas en el banco. Los atributos se especifican mediante las **variables de instancia** de la clase.

Encapsulamiento y ocultamiento de información

Las clases y sus objetos **encapsulan** (envuelven) sus atributos y métodos. Los atributos y métodos de una clase (y sus objetos) están muy relacionados entre sí. Los objetos se pueden comunicar entre sí, pero por lo general no se les permite saber cómo están implementados otros objetos; los detalles de implementación están *ocultos* dentro de los mismos objetos. Este **ocultamiento de información**, como veremos más adelante, es crucial para la buena ingeniería de software.

Herencia

Mediante la **herencia** es posible crear con rapidez y de manera conveniente una nueva clase de objetos. La nueva clase (conocida como **subclase**) comienza con las características de una clase existente (conocida como **superclase**), con la posibilidad de personalizarlas y agregar características únicas propias. En nuestra analogía del auto, sin duda un objeto de la clase “convertible” es un objeto de la clase más *general* llamada “automóvil” pero, de manera más *específica*, el toldo puede ponerse o quitarse.

Interfaces

Java también soporta las **interfaces**: colecciones de métodos relacionados que por lo general nos permiten indicar a los objetos *qué* hacer, pero no *cómo* hacerlo (en Java SE 8 veremos una excepción a esto). En la analogía del auto, una interfaz con “capacidades básicas de conducción” que consista de un volante de dirección, un pedal acelerador y un pedal del freno, permitiría a un conductor indicar al auto *qué* debe hacer. Una vez que sepa cómo usar esta interfaz para dar vuelta, acelerar y frenar, podrá conducir muchos tipos de autos, incluso aunque los fabricantes puedan *implementar* estos sistemas de manera *diferente*.

Una clase **implementa** cero o más interfaces, cada una de las cuales puede tener uno o más métodos, al igual que un auto implementa interfaces separadas para las funciones básicas de conducción, para el control del radio, el control de los sistemas de calefacción y aire acondicionado, etcétera. Así como los fabricantes de automóviles implementan las capacidades de manera *diferente*, las clases pueden implementar los métodos de una interfaz de manera *diferente*. Por ejemplo, un sistema de software puede incluir una interfaz de “respaldo” que ofrezca los métodos *guardar* y *restaurar*. Las clases pueden implementar esos métodos de manera distinta, dependiendo de los tipos de cosas que se vayan a respaldar, como programas, textos, archivos de audio, videos, etc., y los tipos de dispositivos en donde se vayan a almacenar estos elementos.

Análisis y diseño orientado a objetos (A/DOO)

Pronto escribirá programas en Java. ¿Cómo creará el **código** (es decir, las instrucciones) para sus programas? Tal vez, al igual que muchos programadores, sólo encenderá su computadora y empezará a escribir. Quizás este método funcione para pequeños programas (como los que presentamos en los primeros capítulos del

libro), pero ¿qué tal si le pidieran crear un sistema de software para controlar miles de cajeros automáticos para un banco importante? O ¿qué tal si le piden que trabaje con un equipo de 1,000 desarrolladores de software para crear el nuevo sistema de control de tráfico aéreo en Estados Unidos? Para proyectos tan grandes y complejos, no es conveniente tan sólo sentarse y empezar a escribir programas.

Para crear las mejores soluciones, debe seguir un proceso de **análisis** detallado para determinar los **requerimientos** de su proyecto (definir *qué* se supone que debe hacer el sistema) y desarrollar un **diseño** que los satisfaga (decidir *cómo* debe hacerlo el sistema). Lo ideal sería pasar por este proceso y revisar el diseño con cuidado (además de pedir a otros profesionales de software que revisen su diseño) antes de escribir cualquier código. Si este proceso implica analizar y diseñar su sistema desde un punto de vista orientado a objetos, se denomina **proceso de análisis y diseño orientado a objetos** (**A/DOO**). Los lenguajes como Java son orientados a objetos. La programación en un lenguaje de este tipo, conocida como **programación orientada a objetos** (**POO**), le permite implementar un diseño orientado a objetos como un sistema funcional.

El UML (Lenguaje unificado de modelado)

Aunque existen muchos procesos de A/DOO distintos, hay un solo lenguaje gráfico para comunicar los resultados de *cualquier* proceso de A/DOO que se utiliza en la mayoría de los casos. Este lenguaje, conocido como Lenguaje unificado de modelado (UML), es en la actualidad el esquema gráfico más utilizado para modelar sistemas orientados a objetos. Presentamos nuestros primeros diagramas de UML en los capítulos 3 y 4; después los utilizamos en nuestro análisis más detallado de la programación orientada a objetos en el capítulo 11. En nuestro ejemplo práctico *opcional* de ingeniería de software del ATM en los capítulos 33 y 34 presentamos un subconjunto simple de las características del UML, mientras lo guiamos por una experiencia de diseño orientada a objetos.