CAPITULO 3. INTERACCION DE OBJETOS

3.2 Abstracción Y Modularización

Cuando un problema se agranda se vuelve más difícil mantener todos los detalles al mismo tiempo. La solución que usamos para tratar el problema de la complejidad es la *abstracción*; la **abstracción** es la habilidad de ignorar los detalles de las partes para centrar la atención en un nivel más alto de un problema. Dividimos el problema en subproblemas, luego en sub-subproblemas y así sucesivamente, hasta que los problemas resultan suficientemente fáciles de tratar. Una vez que resolvemos uno de los subproblemas no pensamos más sobre los detalles de esa parte, pero tratamos la solución hallada como un bloque de construcción para nuestro siguiente problema. Esta técnica se conoce como la técnica del ***divide y vencerás***.

La **modularización** es el proceso de dividir un todo en partes bien definidas que pueden ser construidas y examinadas separadamente, las cuales interactúan entre si de maneras bien definidas.

La modularización y la abstracción se complementan mutuamente. La modularización es el proceso de dividir cosas grandes (problemas) en partes más pequeñas, mientras que la abstracción es la habilidad de ignorar los detalles para concentrarse en el cuadro más grande.

3.3 Abstracción en Software

En el caso de programas complejos, para mantener una visión global del problema tratamos de identificar los componentes que podemos programar como entidades independientes, y luego intentamos utilizar esos componentes como si fueran partes simples sin tener en cuenta su complejidad interna. En programación orientada a objetos, estos componentes y subcomponentes son objetos.

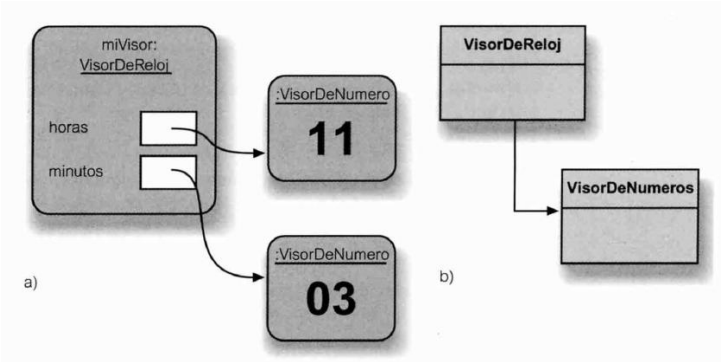
Las clases definen tipos. El nombre de una clase puede ser usado como el tipo de una variable. Las variables cuyo tipo es una clase pueden almacenar objetos de dicha clase.

El tipo de un campo especifica la naturaleza del valor que puede almacenarse en dicho campo. Si el tipo es una clase, el campo puede contener objetos de esa clase.

3.6 Comparación de Diagramas de Clases con Diagramas de Objetos

El **diagrama de clases** muestra las clases de una aplicación y las relaciones entre ellas. Da información sobre el código. Representa la vista estática de un programa.

El **diagrama de objetos** muestra los objetos y sus relaciones en un momento dado de la ejecución de una aplicación. Da información sobre los objetos en tiempo de ejecución. Representa la vista dinámica de un programa.



a) diagrama de objetos; b) diagrama de clases

El diagrama de objetos también muestra otro detalle importante: cuando una variable almacena un objeto, éste no es almacenado directamente en la variable sino que en la variable sólo se almacena una *referencia al objeto.*

Referencia a un objeto. Las variables de tipo objeto almacenan referencias a los objetos.

3.7 Tipos Primitivos y Tipos Objeto

Java reconoce dos clases de tipos muy diferentes: los *tipos primitivos* y los *tipos objeto.* Los tipos primitivos están todos predefinidos en el lenguaje Java

Los **tipos primitivos** en Java son todos los tipos que no son objetos. Los tipos primitivos más comunes son los tipos int, booleano, char, double y long. Los tipos primitivos no poseen métodos.

Tanto los tipos primitivos como los tipos objeto pueden ser usados como tipos, pero existen situaciones en las que se comportan de manera muy diferente. Una diferencia radica en cómo se almacenan los valores. Los valores primitivos se almacenan directamente en una variable. Por otro lado, los objetos no se almacenan directamente en una variable sino que se almacena una referencia al objeto.

Operadores Lógicos

Los operadores lógicos operan con valores booleanos (verdadero o falso) y producen como resultado un nuevo valor booleano. Los tres operadores lógicos más importantes son «y», «o» y «no». En Java se escriben: **&&** (y), || (o) y ! (no).

La expresión **a && b** es verdadera si tanto a como b son verdaderas, en todos los otros casos es falsa.

La expresión **a || b** es verdadera si alguna de las dos es verdadera, puede ser a o puede ser b o pueden ser las dos; si ambas son falsas el resultado es falso.

La expresión **!a** es verdadera si a es falso, y es falsa si a es verdadera.

3.8.2 Concatenación de cadenas

El operador suma (+) tiene diferentes significados dependiendo del tipo de sus operandos. Si ambos operandos son números, el operador + representa la adición. Si los operandos son cadenas, el significado del signo más es la concatenación de cadenas y el resultado es una única cadena compuesta por los dos operandos. Si uno de los operandos del operador más es una cadena y el otro no, el operando que no es cadena es convertido automáticamente en una cadena y luego se realiza la concatenación correspondiente. Esta conversión funciona para todos los tipos. Cualquier tipo que se «sume» con una cadena, automáticamente es convertido a una cadena y luego concatenado.

3.8.3 El Operador Módulo

El operador *módulo* (%). El operador módulo calcula el resto de una división entera. (27%4) será 3.

3.9 Objetos que Crean Objetos

Creación de objetos. Los objetos pueden crear otros objetos usando el operador new. La sintaxis de una operación para crear un objeto nuevo es: new NombreDeClase (lista-de-parámetros).

La operación new hace dos cosas:
1. Crea un nuevo objeto de la clase nombrada.
2. Ejecuta el constructor de dicha cIase.

Si el constructor de la clase tiene parámetros, los parámetros actuales deben ser proporcionados en la sentencia new.

3.10 Constructores Múltiples

Es común que las declaraciones de clases contengan versiones alternativas de constructores o métodos que proporcionan varias maneras de llevar a cabo una tarea en particular mediante diferentes conjuntos de parámetros. Este punto se conoce como ***sobrecarga*** de un constructor o método. Una clase puede contener más de un constructor o más de un método con el mismo nombre, siempre y cuando tengan distintos conjuntos de parámetros que se diferencien por sus tipos.

3.11 Llamadas a Métodos

3.11.1 Llamada a métodos internos

Los métodos pueden llamar a otros métodos de la misma clase como parte de su implementación. Esto se denomina ***llamada* a *método interno***. Se denomina así porque este método está ubicado en la misma clase en que se produce su llamada. Las llamadas a métodos internos tienen la siguiente sintaxis: nombreDelMétodo (lista -de-parámetros). Cuando se encuentra una llamada a un método, se ejecuta este último, y luego de su ejecución se vuelve a la llamada al método y se continúa con la sentencia que sigue a la invocación. Para que la llamada a un método coincida con la signatura del mismo, deben coincidir tanto el nombre del método como su lista de parámetros.

3.11.2 Llamada a métodos externos

Los métodos pueden llamar a métodos de otros objetos usando la notación de punto: se denomina ***llamada* a *método externo***. La sintaxis de una llamada a un método externo es objeto.nombreDelMétodo (lista-de-parámetros). Esta sintaxis se conoce con el nombre de *"notación de punto"*. Es particularmente importante apreciar que usamos aquí el nombre de un objeto y no el nombre de una clase.

El conjunto de métodos de un objeto que está disponible para otros objetos se denomina su ***interfaz***.

3.12 Debugger

Un **depurador** es una herramienta de software que ayuda a examinar cómo se ejecuta una aplicación. Puede usarse para encontrar problemas. Un depurador es un programa que permite que los programadores ejecuten una aplicación paso a paso. Generalmente, ofrece funciones para detener y comenzar la ejecución de un programa en un punto seleccionado del código y para examinar los valores de las variables.

3.12.2 La Palabra Clave this

Observe la siguiente sentencia con la palabra clave this: this.de = de; La línea en su totalidad es una sentencia de asignación, asigna el valor del lado derecho (de) a la variable que está del lado izquierdo (this.de) del símbolo igual (=). El motivo por el que se usa esta construcción radica en que tenemos una situación que se conoce como *sobrecarga de nombres*, y significa que el mismo nombre es usado por entidades diferentes. Es importante comprender que los campos (de un objeto) y los parámetros (o variables locales) son variables que existen independientemente unas de otras, aun cuando compartan nombres similares. Un parámetro (o variable local) y un campo que comparten un nombre no representan un problema para Java. La especificación de Java responde a esta pregunta: Java especifica que siempre se usará la declaración más cercana encerrada en un bloque. Dado que el parámetro (o variable local) de está declarado en el método y el campo de está declarado en la clase, se usará el parámetro (o variable local) pues su declaración es la más cercana a la sentencia que lo usa.

La expresión this hace referencia al objeto actual.