CAPITULO 8. MEJORAR LA ESTRUCTURA MEDIANTE HERENCIA

8.2 Usar Herencia

La **herencia** nos permite definir una clase como una extensión de otra.

Una **superclase** (clase padre/base) es una clase que es extendida por otra clase.

Una **subclase** (clase hija) es una clase que extiende a otra clase. Hereda todos los campos y los métodos de la superclase.

8.3 Jerarquías de herencia

Más de una subclase puede heredar de la misma superclase y una subclase puede convertirse en la superclase de otras subclases. En consecuencia, las clases forman una *jerarquía de herencia*.

Las clases que están vinculadas mediante una relación de herencia forman una **jerarquía de herencia**.

La herencia es una técnica de abstracción que nos permite categorizar las clases de objetos bajo cierto criterio y nos ayuda a especificar las características de estas clases.

8.4 Herencia en Java

La palabra clave extends define la relación de herencia. La frase «extends <SuperClase>» especifica que esta clase es una subclase de la clase SuperClase.

public class <Nombre SubClase> extends <Nombre SuoerClase>
{…}

La subclase define sólo aquellos campos que son únicos para los objetos de su tipo. Los campos de la superclase se heredan y no necesitan ser incluidos en el código de la subclase.

8.4.1 Herencia y derechos de acceso

Los miembros definidos como públicos, ya sea en la superclase o en la subclase, serán accesibles para los objetos de otras clases, pero los miembros definidos como privados serán inaccesibles. En realidad, la regla de privacidad también se aplica entre una subclase y su superclase: una subclase no puede acceder a los miembros privados de su superclase. Se concluye que si un método de una subclase necesita acceder o modificar campos privados de su superclase, entonces la superclase necesitará ofrecer los métodos de acceso y/o métodos de modificación apropiados. Una subclase puede invocar a cualquier método público de su superclase como si fuera propio, no se necesita ninguna variable.

8.4.2 Herencia e inicialización

Cuando creamos un objeto, el constructor de dicho objeto tiene el cuidado de inicializar todos los campos con algún estado razonable.

En primer lugar, la superclase tendrá un constructor aun cuando no tengamos intención de crear, de manera directa, una instancia de la superclase. Este constructor recibe los parámetros necesarios para inicializar los campos de una instancia y contiene el código para llevar a cabo esta inicialización.

El constructor de la subclase recibe los parámetros necesarios para inicializar tanto los campos propios como los de la superclase. El constructor de la subclase contendrá el siguiente código: super(<lista de parámetros>); La palabra clave super es, en realidad, una llamada al constructor de la superclase; el efecto de esta llamada es que se ejecuta el constructor de la supercase, formando parte de la ejecución del constructor de la subclase. Para que esta operación funcione, los parámetros necesarios para la inicialización de los campos de la superclase se pasan a su constructor como parámetros en la llamada a super.

Constructor de superclase. El constructor de una subclase debe tener siempre como primera sentencia una invocación al constructor de su superclase. Si el código no incluye esta llamada, Java intentará insertarla automáticamente. La inserción automática de la llamada a la superclase sólo funciona si la superclase tiene un constructor sin parámetros; en el caso contrario, Java informa un error.

En general, es una buena idea la de incluir siempre en los constructores llamadas explícitas a la superclase, aun cuando sea una llamada que el compilador puede generar automáticamente. Consideramos que esta inclusión forma parte de un buen estilo

de programación, ya que evita la posibilidad de una mala interpretación y de confusión en el caso de que un lector no esté advertido de la generación automática de código.

La **herencia** nos permite reutilizar en un nuevo contexto clases que fueron escritas previamente. El efecto de la reutilización es que se necesita una cantidad menor de código nuevo cuando introducimos elementos adicionales.

Las clases que no se piensan usar para crear instancias, pero cuyo propósito es exclusivamente servir como superclases de otras clases se denominan *clases abstractas*.

8.6 Ventajas de la herencia (hasta ahora)

• Evita la duplicación de código. El uso de la herencia evita la necesidad de escribir copias de código idénticas o muy similares dos veces (o con frecuencia, aún más veces).

• Se reutiliza código. El código que ya existe puede ser reutilizado. Si ya existe una clase similar a la que necesitamos, a veces podemos crear una subclase a partir de esa clase existente y reutilizar un poco de su código en lugar de implementar todo nuevamente.

• Facilita el mantenimiento. El mantenimiento de la aplicación se facilita pues la relación entre las clases está claramente expresada. Un cambio en un campo o en un método compartido entre diferentes tipos de subclases se realiza una sola vez.

• Facilita la extensibilidad. En cierta manera, el uso de la herencia hace mucho más fácil la extensión de una aplicación.

8.7. Subtipos

Subtipo. Por analogía con la jerarquía de clases, los tipos forman una jerarquía de tipos. El tipo que se define mediante la definición de una subclase es un subtipo del tipo de su superclase.

Hasta ahora, hemos interpretado el requerimiento de que los tipos de los parámetros "deben coincidir" como equivalente a decir que "deben ser del mismo tipo": por ejemplo, que el nombre del tipo de un parámetro actual debe ser el mismo que el nombre del tipo del correspondiente parámetro formal. En realidad, esta es sólo una parte de la verdad porque los objetos de las subclases pueden usarse en cualquier lugar que se requiera el tipo de su superclase.

8.7.2 Subtipos y asignación

Variables y subtipos. Las variables pueden contener objetos del tipo declarado o de cualquier subtipo del tipo declarado.

El tipo de una variable declara qué es lo que puede almacenar. La declaración de una variable de tipo <superclase> determina que esta variable puede referenciar instancias de tipo superclase. Pero como una <subclase> también tiene el subtipo <superclase>, es perfectamente legal almacenar una <subclase> en una variable que está pensada para almacenar <superclase>.

Este principio se conoce como ***sustitución***, Se pueden usar objetos de subtipos en cualquier lugar en el que se espera un objeto de un supertipo, ya que el objeto de la subclase es un caso especial de la superclase.

Sin embargo hay que tener en cuenta que a una variable de tipo <subclase> no se le puede asignar un tipo <superclase>, ni una <subclase> diferente.

8.7.3 Subtipos y paso de parámetros

El paso de parámetros (es decir, asignar un parámetro actual a un parámetro formal) se comporta exactamente de la misma manera que la asignación ordinaria a una variable. Este es el motivo por el que podemos pasar un objeto de tipo <subclase> al método que tiene un parámetro de tipo <superclase>.

8.7.4 Variables polimórficas

En Java, las variables que contienen objetos son variables *polimórficas.* El término "polimórfico" se refiere al hecho de que una misma variable puede contener objetos de diferentes tipos (del tipo declarado o de cualquier subtipo del tipo declarado). El polimorfismo aparece en los lenguajes orientados a objetos en numerosos contextos, las variables polimórficas constituyen justamente un primer ejemplo.

La herencia evita la duplicación de código no sólo en las clases servidoras sino también en las clases clientes de aquellas.

8.7.5 Enmascaramiento de tipos

Algunas veces, la regla de que no puede asignarse un supertipo a un subtipo es más restrictiva de lo necesario:

```
Vehiculo v;
Coche a new Coche ();
v = a; / / es correcta
a = v; / / es un error
```

El compilador traduce el código línea por línea, de modo que analiza la última línea aislada de las restantes, sin saber que es lo que realmente se almacena en la variable v. Este problema se denomina *pérdida de tipo*. Podemos resolver este problema diciendo explícitamente al sistema, que la variable v contiene un objeto Coche, y lo hacemos utilizando el operador de enmascaramiento de tipos:

```
a = (Coche) v; / / correcto
```

El operador de enmascaramiento consiste en el nombre de un tipo (en este caso, Coche) escrito entre paréntesis, que precede a una variable o a una expresión. Al usar esta operación, el compilador creerá que el objeto es un Coche y no informará ningún error. Sin embargo, en tiempo de ejecución, el sistema Java verificará si realmente es un Coche. Si fuimos cuidadosos, todo estará bien; si el objeto almacenado en *v* es de otro tipo, el sistema indicará un error en tiempo de ejecución (denominado ClassCastException) y el programa se detendrá.

El enmascaramiento solo se puede hacer con variables que constituyen una relación subtipo/supertipo, sino el compilador produce un error.

El enmascaramiento debiera evitarse siempre que sea posible, porque puede llevar a errores en tiempo de ejecución y esto es algo que claramente no queremos. El compilador no puede ayudarnos a asegurar la corrección de este caso. En la práctica, raramente se necesita del enmascaramiento en un programa orientado a objetos bien estructurado. En la mayoría de los casos, cuando se use un enmascaramiento en el código, debiera reestructurarse el código para evitar el enmascaramiento, y se terminará con un programa mejor diseñado.

8.8 La clase Object

Todas aquellas clases que no tienen una superclase explícita tienen como su superclase a la clase **Object**. Object es una clase de la biblioteca estándar de Java que sirve como superclase para todos los objetos.

El compilador de Java inserta automáticamente la superclase Object en todas las clases que no tengan una declaración explícita extends por lo que jamás es necesario hacer esto manualmente. Cada clase simple (con la única excepción de la clase Object en sí misma) deriva de Object, ya sea directa o indirectamente.

El que todos los objetos tengan una superclase en común tiene dos propósitos. Primero, podemos declarar variables polimórficas de tipo Object que pueden contener cualquier objeto. En segundo lugar, la clase Object puede definir algunos métodos que están automáticamente disponibles para cada objeto existente.

8.9 Autoboxing y clases «envoltorio»

Los tipos primitivos tales como int, boolean y char están separados de los tipos objeto. Sus valores no son instancias de clases y no derivan de la clase Object. Debido a esto, no son suptipos de Object y normalmente, no es posible ubicarlos dentro de una colección.

Este es un inconveniente pues existen situaciones en las que quisiéramos crear, por ejemplo, una lista de enteros (int) o un conjunto de caracteres (char). ¿Qué podemos hacer?

La solución de Java para este problema son las *clases envoltorio.* En Java, cada tipo simple o primitivo tiene su correspondiente clase envoltorio que representa el mismo tipo pero que, en realidad, es un tipo objeto. Por ejemplo, la clase envoltorio para el tipo simple int es la clase de nombre Integer.

El almacenamiento de valores primitivos en un objeto colección se lleva a cabo aún más fácilmente mediante una característica del compilador conocida como *autoboxing.*

En cualquier lugar en el que se use un valor de un tipo primitivo en un contexto que requiere un tipo objeto, el compilador automáticamente envuelve al valor de tipo primitivo en un objeto con el envoltorio adecuado. Esto quiere decir que los valores de tipos primitivos se pueden agregar directamente a una colección.

La operación inversa, *unboxing*, también se lleva a cabo automáticamente, es decir asignar a una variable primitiva un objeto de la clase envoltorio correspondiente.

El proceso de autoboxing se aplica en cualquier lugar en el que se pase como parámetro un tipo primitivo a un método que espera un tipo envoltorio, y cuando un valor primitivo se almacena en una variable de su correspondiente tipo envoltorio. De manera similar, el proceso de unboxing se aplica cuando un valor de tipo envoltorio se pasa como parámetro a un método que espera un valor de tipo primitivo, y cuando se almacena en una variable de tipo primitivo.