CAPITULO 10. MAS TECNICAS DE ABSTRACCION

10.3.2 Métodos abstractos

Determinamos el tipo de una variable mediante el uso del operador instanceof. El operador instanceof evalúa si un objeto determinado es una instancia de determinada clase, directa o indirectamente. La evaluación de la sentencia obj instanceof MiClase da resultado true si el tipo dinámico de obj es MiClase o cualquier subclase de MiClase.

Las clases que no se definen con la intención de crear objetos, sino que solo sirven como superclases, se conocen como ***clases abstractas***. Estas clases pueden contener métodos abstractos. La definición de un **método abstracto** consiste en la signatura de un método sin su correspondiente cuerpo. Se indica mediante la palabra clave abstract.

abstract public <tipoRetorno> <nombreMetodo> (<parametros>);

Un método abstracto se caracteriza por dos detalles:

- Está precedido por la palabra clave abstract.
- No tiene cuerpo y su encabezado termina con un punto y coma.

Dado que el método no tiene cuerpo, jamás podrá ser ejecutado.

10.3.3 Clases abstractas

No sólo los métodos pueden declararse como abstractos, también las clases pueden declararse abstractas. Las clases se declaran abstractas mediante la inserción de la palabra clave abstract en el encabezado de la clase.

public abstract class <nombreClase>{…}

Una **clase abstracta** es una clase de la que no se pretende crear instancias. Su propósito es servir como superclase a otras clases. Solo las clases abstractas pueden contener métodos abstractos.

Las clases que no son abstractas (todas las clases que hemos visto previamente) se denominan ***clases concretas***.

La declaración de una clase abstracta sirve a varios propósitos:

- No se creará ninguna instancia de clases abstractas. El intento de uso de la palabra clave new con una clase abstracta genera un error. Establecemos que no deseamos instancias creadas directamente a partir de la clase abstracta: esta clase sólo sirve como superclase. La declaración de la clase como abstracta refuerza esta restricción.
- Sólo las clases abstractas pueden tener métodos abstractos. Esto asegura que siempre podrán ser ejecutados todos los métodos de las clases concretas. Si permitiéramos un método abstracto en una clase concreta, podríamos crear una instancia de una clase a la que le falta la implementación de un método.
- Las clases abstractas con métodos abstractos fuerzan a las subclases a sobrescribir una implementación de aquellos métodos declarados abstractos. Si una subclase no provee una implementación para un método abstracto heredado, es en sí misma abstracta, y no puede crearse ninguna instancia. Para que una subclase sea concreta, debe proveer implementaciones para *todos* los métodos abstractos heredados.

Ahora podemos comenzar a ver el propósito de los métodos abstractos. Aunque no proveen una implementación, aseguran que todas las subclases tienen una implementación de este método.

Subclases abstractas. Para que una subclase de una clase abstracta se convierta en una subclase concreta, debe proveer las implementaciones de todos los métodos abstractos heredados. De lo contrario, será propiamente abstracta.

10.4 Más métodos abstractos

En Java, los campos se manejan de manera diferente que los métodos: los campos no pueden ser sobrescritos por las versiones de las subclases, independientemente de si un campo es estático o no.

10.5 Herencia múltiple

Herencia múltiple. Una situación en la que una clase deriva directamente de más de una superclase se denomina herencia múltiple. En consecuencia, la subclase tiene todas las características de ambas superclases y aquellas definidas en la subclase propiamente dicha.

La herencia múltiple es en principio, muy fácil de comprender pero su implementación en un lenguaje de programación puede llegar a ser significativamente complicada. Los diferentes lenguajes orientados a objetos varían en cuanto a su tratamiento de la herencia múltiple: algunos lenguajes permiten la herencia múltiple de superclases y otros no. Java se encuentra en un lugar intermedio: no permite la herencia múltiple de clases pero proporciona otra construcción denominada «interfaces» que permite una forma limitada de herencia múltiple.

10.6 Interfaces

Java captura este concepto más formalmente permitiendo definir los *tipos interfaces*. En una primer mirada, las interfaces son similares a las clases, la diferencia más obvia radica en que sus definiciones de métodos no incluyen cuerpos. Por lo tanto, se parecen a las clases abstractas en las que todos sus métodos son abstractos.

Una interfaz en Java es la especificación de un tipo (bajo la forma de un nombre de tipo y un conjunto de métodos) que no define ninguna implementación para los métodos.

Las interfaces en Java tienen una cantidad de características importantes:

- En el encabezado de la declaración se usa la palabra clave interface en lugar de class.
- Todos los métodos de una interfaz son abstractos; no se permiten métodos con cuerpo. No es necesaria la palabra clave abstract.
- Las interfaces no contienen ningún constructor.
- Todas las signaturas de los métodos de una interfaz tienen visibilidad pública. No es necesario declarar la visibilidad: por ejemplo, no es necesario que cada método contenga la palabra clave public.
- En una interfaz, sólo se permiten los campos constantes (campo público, estático y final). Pueden omitirse las palabras clave public, static y final pero todos los campos, igualmente, serán tratados como públicos, estáticos y finales.

Una clase puede derivar de una interfaz de la misma manera en que deriva de una clase. Sin embargo, Java utiliza una palabra clave diferente, implements, para la herencia a partir de interfaces. Se dice que una clase *implementa* una interfaz si incluye una *cláusula implements* en su encabezado.

public class <miClase> implements <interfaz1>{…}

10.6.2 Herencia de múltiples interfaces

Como mencionamos anteriormente, Java permite que cada clase extienda como máximo a otra clase, sin embargo permite que una clase implemente cualquier número de interfaces (además de la posibilidad de extender una clase).

public class <miClase> extends <superClase> implements <interfaz1>, <interfaz2>, <interfaz3>{…}

La clase que imprenta interfaces hereda las definiciones de los métodos de todas las interfaces como métodos abstractos. En consecuencia, se deben proveer definiciones para los métodos sobrescribiendo los métodos, de lo contrario la clase se declara abstracta. Si una clase no provee un cuerpo para un método de las interfaces que implementa, es propiamente abstracta (y debe incluir la palabra clave abstract en su encabezado).

10.6.3 Interfaces como tipos

Cuando una clase implementa una interfaz no hereda ninguna implementación de ella, pues las interfaces no pueden contener cuerpos de métodos. Entonces, la pregunta que cabe es: ¿qué ganamos realmente al implementar interfaces? Cuando presentamos la herencia pusimos énfasis en dos grandes beneficios de la herencia:

- La subclase hereda el código (la implementación de métodos y campos) de la superclase. Esto permite la reutilización de código existente y evita la duplicación de código.
- La subclase se convierte en un subtipo de la superclase. Esto permite la existencia de variables polimórficas y la invocación polimórfica de métodos. En otras palabras, permite que los casos especiales de objetos (instancias de subclases) se traten de manera uniforme (como instancias del supertipo).

Las interfaces no brindan el primer beneficio (ya que no contienen ninguna implementación), pero sí ofrecen el segundo. Una interfaz define un tipo tal como lo hace una clase. Esto quiere decir que las variables pueden ser declaradas del tipo de la interfaz, aun cuando no pueda existir ningún objeto de tal tipo (sólo de los subtipos).

Las interfaces no pueden tener instancias directas pero sirven como supertipos para las instancias de otras clases.

10.6.4 Interfaces como especificaciones

En este capítulo hemos introducido las interfaces con el sentido de implementar herencia múltiple en Java. Este es un uso importante de las interfaces, pero existen otros. La característica más importante de las interfaces es que separan completamente la definición de la funcionalidad (la clase “interfaz” en el sentido más amplio de la palabra) de su implementación. Un buen ejemplo de cómo pueden usarse las interfaces en la práctica se puede encontrar en la jerarquía de las colecciones de Java.

La jerarquía de colecciones define, entre otros tipos, la interfaz List y las clases ArrayList y LinkedList. La interfaz List especifica la funcionalidad total de una lista sin aportar ninguna implementación. Las subclases LinkedList y ArrayList proveen dos implementaciones diferentes para la misma interfaz. Esto es interesante porque las dos implementaciones difieren

enormemente en la eficiencia de algunas de sus funciones. Por ejemplo, el acceso aleatorio de elementos situados en el medio de una lista es mucho más rápido en el ArrayList, sin embargo la inserción o la eliminación de elementos puede ser mucho más rápida en la LinkedList.

La decisión de cuál de las implementaciones resulta mejor para una aplicación determinada puede ser difícil de juzgar anticipadamente, depende mucho de la frecuencia relativa con que se lleven a cabo ciertas operaciones y algunos otros factores.

En la práctica, la mejor forma de descubrir cuál es la mejor es probando: implementar la aplicación con ambas alternativas y medir el rendimiento. La existencia de la interfaz List facilita esta prueba. Si en lugar de usar un ArrayList o una LinkedList como tipo de variable y tipo de parámetro usamos siempre List, nuestra aplicación funcionará independientemente del tipo específico de lista que estemos usando realmente. Debemos usar el nombre específico de la implementación seleccionada sólo cuando creamos una nueva lista. Por ejemplo, podemos escribir private List<Tipo> miLista = new ArrayList<Tipo> ();

Observe que el tipo del campo es justamente List de Tipo. De esta manera, podemos modificar toda la aplicación para que use una lista enlazada con sólo cambiar ArrayList por LinkedList en un único lugar: el lugar en el que se crea la lista.

10.6.6 ¿Clase abstracta o interfaz?

En algunas situaciones se tiene que elegir entre usar una clase abstracta o una interfaz. Algunas veces la elección es fácil: cuando se pretende que la clase contenga implementaciones para algunos métodos necesitamos usar una clase abstracta. En otros casos, tanto la clase abstracta como la interfaz pueden hacer el mismo trabajo. Si tenemos que elegir, es preferible usar interfaces. Si proveemos un tipo mediante una clase abstracta, las subclases no pueden extender ninguna otra clase; dado que las interfaces permiten la herencia múltiple, el uso de una interfaz no crea tal restricción. Por lo tanto, el uso de interfaces da por resultado una estructura más flexible y más extensible.

10.7 Resumen de herencia

En general, distinguimos dos propósitos principales del uso de la herencia: podemos usarla para heredar código (código heredado) y podemos usarla para heredar el tipo (subtipeado). El primero es útil para reutilizar código, el segundo para el polimorfismo y la especialización.

Cuando heredamos a partir de clases concretas (“extends”) hacemos dos cosas: heredamos la implementación y el tipo.

Cuando heredamos a partir de interfaces (“implements”) heredamos un tipo pero no la implementación.

Para los casos en que ambas partes sean útiles podemos heredar a partir de clases abstractas; aquí, heredamos el tipo y una implementación parcial.

Cuando heredamos una implementación completa, podemos elegir agregar o sobrescribir métodos.

Cuando no se hereda ninguna implementación de un tipo o se hereda parcialmente la implementación de un tipo, la subclase debe proveer la implementación antes de que pueda ser instanciada.

Algunos otros lenguajes orientados a objetos también proporcionan mecanismos para heredar código sin heredar el tipo. Java no provee este tipo de construcciones.