Implementando una interfaz

Definición de la interfaz relacionable

Para declarar una clase que implementa una interfaz, se incluye una cláusula *implements* en la declaración de la clase. La clase puede implementar más de una interfaz, por lo que la palabra clave *implements* va seguida de una lista separada por comas de las interfaces implementadas por la clase. Por convención, la cláusula *implements* sigue a la cláusula *extends*, si la hay.

Considere una interfaz que define cómo comparar el tamaño de los objetos.

```
1  public interface Relatable {
2
3      // this (object calling isLargerThan())
4      // and other must be instances of
5      // the same class returns 1, 0, -1
6      // if this is greater than,
7      // equal to, or less than other
8      public int isLargerThan(Relatable other);
9  }
```

Si desea poder comparar el tamaño de objetos similares, sin importar cuáles sean, la clase que los instancia debe implementar *Relatable*.

Cualquier clase puede implementar *Relatable* si existe alguna manera de comparar el "tamaño" relativo de los objetos instanciados a partir de la clase. Para cadenas, podría ser la cantidad de caracteres; para libros, podría ser la cantidad de páginas; para estudiantes, podría ser el peso; y así sucesivamente. Para objetos geométricos planos, el área sería una buena opción (consulte la clase que sigue *RectanglePlus*), mientras que el volumen funcionaría para objetos geométricos tridimensionales. Todas estas clases pueden implementar el método *isLargerThan*().

Si sabe que una clase implementa *Relatable*, entonces sabe que puede comparar el tamaño de los objetos instanciados desde esa clase.

Implementación de la interfaz relacionable

Aquí está la clase *Rectangle* que se presentó en la sección Creación de objetos, reescrita para implementar *Relatable*.

```
public class RectanglePlus
    implements Relatable {
    public int width = 0;
    public int height = 0;
    public Point origin;

    // four constructors
    public RectanglePlus() {
        origin = new Point(0, 0);
    }
    public RectanglePlus(Point p) {
        origin = p;
    }
    public RectanglePlus(int w, int h) {
        origin = new Point(0, 0);
        width = w;
        height = h;
    }
    public RectanglePlus(Point p, int w, int h) {
        origin = p;
        width = w;
        height = h;
    }

    // a method for moving the rectangle
    public void move(int x, int y) {
        origin.x = x;
        origin.y = y;
    }

    // a method for computing
    // the area of the rectangle
    public int getArea() {
        return width * height;
    }

    // a method required to implement
    // the Relatable interface
    public int isLargerThan(Relatable other) {
        RectanglePlus otherRect
            = (RectanglePlus)other;
        if (this.getArea() < otherRect.getArea())
            return -1;
        else if (this.getArea() > otherRect.getArea())
            return 1;
        else
            return 0;
    }
}
```

Debido a que *RectanglePlus* implementa *Relatable*, se puede comparar el tamaño de dos objetos *RectanglePlus* cualquiera.

***Nota**: El método isLargerThan(), tal como se define en la interfaz Relatable, toma un objeto de tipo Relatable. La línea de código convierte **other** en una instancia de RectanglePlus. La conversión de tipo le indica al compilador cuál es realmente el objeto. La invocación directa de getArea() en la instancia **other** (**other**.getArea()) no se compilaría porque el compilador no entiende que **other** es en realidad una instancia de RectanglePlus.*

Interfaces en evolución

Considere una interfaz que ha desarrollado llamada DoIt:

```
public interface DoIt {
   void doSomething(int i, double x);
   int doSomethingElse(String s);
}
```

Supongamos que, en un momento posterior, desea agregar un tercer método a *DoIt*, de modo que la interfaz ahora se convierte en:

```
public interface DoIt {

   void doSomething(int i, double x);
   int doSomethingElse(String s);
   boolean didItWork(int i, double x, String s);

}
```

Si se realiza este cambio, todas las clases que implementan la interfaz *DoIt* anterior dejarán de funcionar porque ya no la implementan. Los programadores que dependen de esta interfaz protestarán en voz alta.

Intente anticipar todos los usos de su interfaz y especifíquela completamente desde el principio. Si desea agregar métodos adicionales a una interfaz, tiene varias opciones. Puede crear una interfaz *DoItPlus* que extienda *DoIt*:

```
public interface DoItPlus extends DoIt {

   boolean didItWork(int i, double x, String s);

}
```

Ahora los usuarios de su código pueden elegir continuar usando la interfaz antigua o actualizar a la nueva interfaz.

Como alternativa, puede definir sus nuevos métodos como métodos predeterminados. El siguiente ejemplo define un método predeterminado denominado *didItWork*():

```java
public interface DoIt {

   void doSomething(int i, double x);
   int doSomethingElse(String s);
   default boolean didItWork(int i, double x, String s) {
       // Method body
   }
}
```

Tenga en cuenta que debe proporcionar una implementación para los métodos predeterminados. También puede definir nuevos métodos estáticos para las interfaces existentes. Los usuarios que tienen clases que implementan interfaces mejoradas con nuevos métodos predeterminados o estáticos no tienen que modificarlas ni volver a compilarlas para incluir los métodos adicionales.

Métodos predeterminados

En la sección Interfaces se describe un ejemplo que involucra a fabricantes de automóviles controlados por computadora que publican interfaces estándar de la industria que describen qué métodos se pueden invocar para operar sus automóviles. ¿Qué sucedería si esos fabricantes de automóviles controlados por computadora añadieran nuevas funciones, como el vuelo, a sus automóviles? Estos fabricantes tendrían que especificar nuevos métodos para permitir que otras empresas (como los fabricantes de instrumentos de guía electrónica) adapten su software a los automóviles voladores. ¿Dónde declararían estos fabricantes de automóviles estos nuevos métodos relacionados con el vuelo? Si los añaden a sus interfaces originales, los programadores que hayan implementado esas interfaces tendrían que reescribir sus implementaciones. Si los añaden como métodos estáticos, los programadores los considerarían métodos de utilidad, no métodos esenciales y básicos.

Los métodos predeterminados le permiten agregar nueva funcionalidad a las interfaces de sus bibliotecas y garantizar la compatibilidad binaria con el código escrito para versiones anteriores de esas interfaces.

Considere la siguiente interfaz *TimeClient*:

```java
import java.time.*;

public interface TimeClient {
    void setTime(int hour, int minute, int second);
    void setDate(int day, int month, int year);
    void setDateAndTime(int day, int month, int year,
                                int hour, int minute, int second);
    LocalDateTime getLocalDateTime();
}
```

La siguiente clase, *SimpleTimeClient*, implementa *TimeClient*:

```java
public class SimpleTimeClient implements TimeClient {

    private LocalDateTime dateAndTime;

    public SimpleTimeClient() {
        dateAndTime = LocalDateTime.now();
    }

    public void setTime(int hour, int minute, int second) {
        LocalDate currentDate = LocalDate.from(dateAndTime);
        LocalTime timeToSet = LocalTime.of(hour, minute, second);
        dateAndTime = LocalDateTime.of(currentDate, timeToSet);
    }

    public void setDate(int day, int month, int year) {
        LocalDate dateToSet = LocalDate.of(day, month, year);
        LocalTime currentTime = LocalTime.from(dateAndTime);
        dateAndTime = LocalDateTime.of(dateToSet, currentTime);
    }

    public void setDateAndTime(int day, int month, int year,
                               int hour, int minute, int second) {
        LocalDate dateToSet = LocalDate.of(day, month, year);
        LocalTime timeToSet = LocalTime.of(hour, minute, second);
        dateAndTime = LocalDateTime.of(dateToSet, timeToSet);
    }

    public LocalDateTime getLocalDateTime() {
        return dateAndTime;
    }

    public String toString() {
        return dateAndTime.toString();
    }

    public static void main(String... args) {
        TimeClient myTimeClient = new SimpleTimeClient();
        System.out.println(myTimeClient.toString());
    }
}
```

Supongamos que desea agregar nueva funcionalidad a la interfaz *TimeClient*, como la capacidad de especificar una zona horaria a través de un objeto *ZonedDateTime*(que es como un objeto *LocalDateTime* excepto que almacena información de zona horaria):

```java
public interface TimeClient {
    void setTime(int hour, int minute, int second);
    void setDate(int day, int month, int year);
    void setDateAndTime(int day, int month, int year,
        int hour, int minute, int second);
    LocalDateTime getLocalDateTime();
    ZonedDateTime getZonedDateTime(String zoneString);
}
```

Después de esta modificación de la interfaz *TimeClient*, también tendría que modificar la clase *SimpleTimeClient* e implementar el método *getZonedDateTime*(). Sin embargo, en lugar de

dejarlo como abstracto *getZonedDateTime*() (como en el ejemplo anterior), puede definir una implementación predeterminada. (Recuerde que un método abstracto es un método declarado sin una implementación).

```
public interface TimeClient {
    void setTime(int hour, int minute, int second);
    void setDate(int day, int month, int year);
    void setDateAndTime(int day, int month, int year,
                                int hour, int minute, int second);
    LocalDateTime getLocalDateTime();

    static ZoneId getZoneId (String zoneString) {
        try {
            return ZoneId.of(zoneString);
        } catch (DateTimeException e) {
            System.err.println("Invalid time zone: " + zoneString +
                "; using default time zone instead.");
            return ZoneId.systemDefault();
        }
    }

    default ZonedDateTime getZonedDateTime(String zoneString) {
        return ZonedDateTime.of(getLocalDateTime(), getZoneId(zoneString));
    }
}
```

Se especifica que una definición de método en una interfaz es un método predeterminado con la palabra clave *default* al comienzo de la firma del método. Todas las declaraciones de métodos en una interfaz, incluidos los métodos predeterminados, son implícitamente públicas, por lo que se puede omitir el modificador público.

Con esta interfaz, no es necesario modificar la clase *SimpleTimeClient*, y esta clase (y cualquier clase que implemente la interfaz *TimeClient*) tendrá el método *getZonedDateTime*() ya definido. El siguiente ejemplo, *TestSimpleTimeClient*, invoca el método *getZonedDateTime*() desde una instancia de *SimpleTimeClient*:

```
public class TestSimpleTimeClient {
    public static void main(String... args) {
        TimeClient myTimeClient = new SimpleTimeClient();
        System.out.println("Current time: " + myTimeClient.toString());
        System.out.println("Time in California: " +
            myTimeClient.getZonedDateTime("Blah blah").toString());
    }
}
```

Ampliación de interfaces que contienen métodos predeterminados

Al extender una interfaz que contiene un método predeterminado, puede hacer lo siguiente:

- No menciona en absoluto el método predeterminado, lo que permite que su interfaz extendida herede el método predeterminado.

- Redeclara el método predeterminado, lo que lo hace abstracto.
- Redefinir el método predeterminado, que lo anula.
- Supongamos que amplía la interfaz *TimeClient* de la siguiente manera:

```
public interface AnotherTimeClient extends TimeClient { }
```

Cualquier clase que implemente la interfaz *AnotherTimeClient* tendrá la implementación especificada por el método predeterminado *TimeClient.getZonedDateTime*().

Supongamos que amplía la interfaz *TimeClient* de la siguiente manera:

```
public interface AbstractZoneTimeClient extends TimeClient {
    public ZonedDateTime getZonedDateTime(String zoneString);
}
```

Cualquier clase que implemente la interfaz *AbstractZoneTimeClient* tendrá que implementar el método *getZonedDateTime*(); este método es un método abstracto como todos los demás métodos no predeterminados (y no estáticos) en una interfaz.

Supongamos que amplía la interfaz *TimeClient* de la siguiente manera:

```
public interface HandleInvalidTimeZoneClient extends TimeClient {
    default public ZonedDateTime getZonedDateTime(String zoneString) {
        try {
            return ZonedDateTime.of(getLocalDateTime(),ZoneId.of(zoneString));
        } catch (DateTimeException e) {
            System.err.println("Invalid zone ID: " + zoneString +
                "; using the default time zone instead.");
            return ZonedDateTime.of(getLocalDateTime(),ZoneId.systemDefault());
        }
    }
}
```

Cualquier clase que implemente la interfaz *HandleInvalidTimeZoneClient* utilizará la implementación *getZonedDateTime*() especificada por esta interfaz en lugar de la especificada por la interfaz *TimeClient*.

Métodos estáticos

Además de los métodos predeterminados, puede definir métodos estáticos en las interfaces. (Un método estático es un método que está asociado con la clase en la que está definido en lugar de con cualquier objeto. Cada instancia de la clase comparte sus métodos estáticos). Esto le facilita la organización de los métodos auxiliares en sus bibliotecas; puede mantener los métodos estáticos específicos de una interfaz en la misma interfaz en lugar de en una clase separada. El siguiente ejemplo define un método estático que recupera un objeto *ZoneId* correspondiente a un identificador de zona horaria; utiliza la zona horaria predeterminada del sistema si no hay ningún objeto *ZoneId* correspondiente al identificador dado. (Como resultado, puede simplificar el método *getZonedDateTime*()):

```
public interface TimeClient {
    // ...
    static public ZoneId getZoneId (String zoneString) {
        try {
            return ZoneId.of(zoneString);
        } catch (DateTimeException e) {
            System.err.println("Invalid time zone: " + zoneString +
                "; using default time zone instead.");
            return ZoneId.systemDefault();
        }
    }

    default public ZonedDateTime getZonedDateTime(String zoneString) {
        return ZonedDateTime.of(getLocalDateTime(), getZoneId(zoneString));
    }
}
```

Al igual que los métodos estáticos en las clases, se especifica que una definición de método en una interfaz es un método estático con la palabra clave *static* al comienzo de la firma del método. Todas las declaraciones de métodos en una interfaz, incluidos los métodos estáticos, son implícitamente públicas, por lo que se puede omitir el modificador público.

A partir de Java SE 9, puede definir métodos privados en una interfaz para abstraer un fragmento de código común de los métodos predeterminados de una interfaz mientras define su implementación. Estos métodos pertenecen a la implementación y no pueden ser predeterminados ni abstractos cuando se definen. Por ejemplo, puede hacer que un método *getZoneId()* sea privado ya que aloja un fragmento de código que es interno a la implementación de la interfaz.

Utilizar una interfaz como tipo

Utilizar una interfaz como tipo

Cuando se define una nueva interfaz, se define un nuevo tipo de datos de referencia. Se pueden utilizar nombres de interfaz en cualquier lugar en el que se pueda utilizar cualquier otro nombre de tipo de datos. Si se define una variable de referencia cuyo tipo es una interfaz, cualquier objeto que se le asigne debe ser una instancia de una clase que implemente la interfaz.

A modo de ejemplo, aquí se muestra un método para encontrar el objeto más grande en un par de objetos, para cualquier objeto que se instancia desde una clase que implementa *Relatable*:

```
public Object findLargest(Object object1, Object object2) {
   Relatable obj1 = (Relatable)object1;
   Relatable obj2 = (Relatable)object2;
   if ((obj1).isLargerThan(obj2) > 0)
      return object1;
   else
      return object2;
}
```

Al convertir object1a un tipo, *Relatable* se puede invocar el método *isLargerThan()*.

Si se desea implementar *Relatable* en una amplia variedad de clases, los objetos instanciados de cualquiera de esas clases se pueden comparar con el método *findLargest*(), siempre que ambos objetos sean de la misma clase. De manera similar, todos se pueden comparar con los siguientes métodos:

```
public Object findSmallest(Object object1, Object object2) {
   Relatable obj1 = (Relatable)object1;
   Relatable obj2 = (Relatable)object2;
   if ((obj1).isLargerThan(obj2) < 0)
      return object1;
   else
      return object2;
}

public boolean isEqual(Object object1, Object object2) {
   Relatable obj1 = (Relatable)object1;
   Relatable obj2 = (Relatable)object2;
   if ( (obj1).isLargerThan(obj2) == 0)
      return true;
   else
      return false;
}
```

Estos métodos funcionan para cualquier objeto "relacionable", sin importar cuál sea su herencia de clase. Cuando implementan Relatable, pueden ser tanto de su propio tipo de clase (o superclase) como de un tipo Relatable. Esto les brinda algunas de las ventajas de la herencia múltiple, donde pueden tener el comportamiento tanto de una superclase como de una interfaz.

Bibliografía: https://dev.java/learn/interfaces/examples/