Programación modular (II)

Ricardo Pérez López

IES Doñana, curso 2020/2021

Generado el 30 de abril de 2021 a las 18:28:00

Índice general

1. Interfaces

1.1. Concepto de interfaz

Hay situaciones en ingeniería del software en las que es importante que grupos dispares de programadores acuerden un «contrato» que explique cómo interactúa su software.

Cada grupo debería poder escribir su código sin ningún conocimiento de cómo se escribe el código del otro grupo.

Ya hemos estudiado que, en esas situaciones, lo que cada grupo crea son **módulos**, y lo que un grupo debe conocer del módulo del otro es su **interfaz**.

Por tanto, en términos generales, las interfaces son tales contratos.

En Java:

- Las clases representan los módulos de los que se compone el programa.

- Una interfaz de Java es una descripción del contrato que debe cumplir una clase.

Las interfaces de Java nos permiten desacoplar (separar) la interfaz de una clase de sus detalles de implementación.

De esta forma, el usuario de una clase no tiene por qué hablar directamente con la clase, sino que lo hace a través de la interfaz que implementa.

Eso permite cambiar una implementación por otra cuando sea necesario, simplemente cambiando la clase que implementa la interfaz por otra diferente, sin que los usuarios de la interfaz se vean afectados.

Por ejemplo, imagine una sociedad futurista en la que automóviles robóticos controlados por computadora transporten pasajeros por las calles de la ciudad sin un operador humano.

Los fabricantes de automóviles escriben software que hace funcionar el automóvil: detener, arrancar, acelerar, girar a la izquierda, etc.

Otro grupo industrial, los fabricantes de instrumentos de guía electrónica, fabrica sistemas informáticos que reciben datos de posición GPS y transmisiones inalámbricas de las condiciones del tráfico, y utilizan esa información para conducir el automóvil.

Los fabricantes de automóviles deben publicar una interfaz estándar que explique en detalle qué métodos pueden invocarse para hacer que el automóvil se mueva (cualquier automóvil, de cualquier fabricante).

Los fabricantes de guías pueden entonces escribir un software que invoque los métodos descritos en la interfaz para controlar el automóvil.

Ningún grupo industrial necesita saber cómo se implementa el software del otro grupo.

De hecho, cada grupo considera que su software es altamente propietario y se reserva el derecho de modificarlo en cualquier momento, siempre que continúe adhiriéndose a la interfaz publicada.

1.2. Definición de interfaces

En Java, una **interfaz** es un tipo referencia, similar a una clase abstracta, que sólo puede contener constantes estáticas, métodos abstractos, métodos predeterminados, métodos estáticos y tipos anidados.

Los únicos métodos *concretos* (es decir, con cuerpo) que puede tener una interfaz son los métodos predeterminados y los métodos estáticos.

No se pueden crear instancias de interfaces; solo pueden implementarse mediante clases o ser usadas como supertipos de otras interfaces.

La definición de una interfaz tiene una sintaxis similar a la de una clase:

```
⟨interfaz⟩ ::= [public] interface ⟨nombre⟩ [extends ⟨superinterfaces⟩] {
        ⟨miembro_interfaz⟩*
}

⟨nombre⟩ ::= identificador
⟨superinterfaces⟩ ::= ⟨superinterfaz⟩[, ⟨superinterfaz⟩]*
```

⟨*superinterfaz*⟩ ::= [⟨*paquete*⟩**.**]**identificador**
⟨*miembro_interfaz*⟩ ::= ⟨*constante*⟩ | ⟨*método_interfaz*⟩
⟨*constante*⟩ ::= [**public**] [**static**] [**final**] ⟨*decl_constantes*⟩
⟨*decl_constantes*⟩ ::= ⟨*tipo*⟩ ⟨*decl_constante*⟩ (**,** ⟨*decl_constante*⟩)* **;**
⟨*decl_constante*⟩ ::= **identificador** ⟨*inic_variable*⟩
⟨*método_interfaz*⟩ ::= ⟨*método_abstracto_interfaz*⟩ | ⟨*método_concreto_interfaz*⟩
⟨*método_abstracto_interfaz*⟩ ::= [**public**] [**abstract**] ⟨*decl_método*⟩
⟨*método_concreto_interfaz*⟩ ::= [**public**] [**default** | **static**] ⟨*def_método*⟩

Como las clases, las interfaces pueden tener dos visibilidades:

- Predeterminada: es visible sólo dentro del paquete donde se ha definido.
- Pública: es visible desde cualquier paquete.

El cuerpo de una interfaz puede contener métodos abstractos, métodos predeterminados y métodos estáticos.

Un método abstracto dentro de una interfaz no tiene cuerpo: es una declaración que acaba en `;`.

Los métodos predeterminados se definen con el modificador `default` y sí tienen cuerpo.

Los métodos estáticos llevan el modificador `static` y también tienen cuerpo.

Todos los métodos abstractos, predeterminados y estáticos de una interfaz son implícitamente públicos, por lo que se puede omitir el modificador `public`.

Todas las constantes de una interfaz son implícitamente `public`, `final` y `static`, por lo que se pueden omitir todos esos modificadores.

Por ejemplo, podemos definir una interfaz `Sumable` que represente cualquier cosa que se pueda sumar:

```
interface Sumable {
    Sumable sumar(Sumable otro);
}
```

O una interfaz `Comparable` que represente cualquier cosa que se pueda comparar con otra para ver si es menor que ella:

```
interface Comparable {
    boolean menorQue(Comparable otro);
}
```

1.3. Implementación de interfaces

Para usar una interfaz, se necesita una clase que implemente esa interfaz.

Para ello, la definición de la clase debe llevar la cláusula `implements` indicando las interfaces que implementa dicha clase.

Además, las clases que deseen implementar esa interfaz, deben implementar (es decir, proporcionar un cuerpo) a todos los métodos abstractos declarados en la interfaz.

En la implementación del método se recomienda (pero no es necesario) usar el decorador `@Override`.

Por supuesto, una clase puede tener una superclase y, al mismo tiempo, implementar una o varias interfaces.

Por ejemplo:

```
class Numero implements Sumable {
    int num;

    Numero(int num) {
        this.num = num;
    }

    @Override
    public Sumable sumar(Sumable otro) {
        Numero otroNumero = (Numero) otro;
        return new Numero(num + otroNumero.num);
    }
}
```

Podemos hacer que una clase implemente tantas interfaces como sea necesario:

```
class Numero implements Sumable {
    public int num;

    Numero(int num) {
        this.num = num;
    }

    @Override
    public Numero sumar(Sumable otro) {
        Numero otroNumero = (Numero) otro;
        return new Numero(num + otroNumero.num);
    }

    @Override
    public boolean menorQue(Comparable otro) {
        Numero otroNumero = (Numero) otro;
        return num < otroNumero.num;
    }
}
```

1.4. Las interfaces como tipos

Las interfaces son tipos en Java.

Por tanto, ya hemos visto tres entidades distintas que representan tipos en Java:

- Las clases
- Los *arrays*
- Las interfaces.

Por tanto, el nombre de una interfaz se puede usar allí donde se espere el nombre de un tipo.

Eso significa que podemos declarar variables, parámetros, métodos, etc. cuyo tipo es una interfaz.

Cuando una clase implementa una interfaz, esa clase se convierte en subtipo directo del tipo que representa la interfaz:

Si `C` implementa `I`, entonces $\texttt{C} <_1 \texttt{I}$.

Cuando algo se declara con un tipo representado mediante una interfaz, el valor de ese algo debe ser una instancia de una clase que implemente esa interfaz.

Dicho de otra forma: allí donde se espere un valor de un tipo interfaz, se puede poner una instancia de una clase que implemente esa interfaz.

Eso significa que un objeto puede tener muchos tipos:

- El tipo de su propia clase (la que se usó para instanciar el objeto).
- El tipo de todas sus superclases.
- El tipo de todas las interfaces que implementa su propia clase.
- El tipo de todas las interfaces que implementa todas sus superclases.
- El tipo de todas las superinterfaces de todas las interfaces que implementan su propia clase y todas sus superclases (ya lo veremos).

Por ejemplo, allí donde se espere un valor de tipo `Sumable`, se puede poner una instancia de la clase `Numero`, ya que la clase `Numero` implementa la interfaz `Sumable`:

```
public class Interfaces {
    public static void main(String[] args) {
        Numero n1 = new Numero(4);
        Numero n2 = new Numero(5);
        Numero resultado = n1.sumar(n2);

        System.out.println(resultado.num);     // Imprime «9»
    }
}
```

El parámetro del método `sumar` está declarado de tipo `Sumable`, pero en el código anterior le hemos enviado un argumento de tipo `Numero`.

Esto es perfectamente válido, ya que `Numero <: Sumable` y, por el principio de sustitución, podemos enviar un valor de un subtipo del tipo declarado para el parámetro.

También podríamos haber hecho:

```
public class Interfaces {
    public static void main(String[] args) {
        Sumable n1 = new Numero(4);
        Sumable n2 = new Numero(5);
        Sumable resultado = n1.sumar(n2);

        System.out.println(resultado.num);     // Imprime «9»
    }
}
```

Es decir: usar `Sumable` en lugar de `Numero` como tipo estático de las variables `n1`, `n2` y `resultado`.

1.5. Herencia entre interfaces

Es posible establecer relaciones de generalización entre interfaces.

En tal caso, podemos hablar de subinterfaces y superinterfaces.

En Java, la generalización entre interfaces es múltiple, lo que significa que una interfaz puede tener más de una superinterfaz directa.

Las superinterfaces directas de una interfaz se indican en la definición de la subinterfaz usando la clásula **`extends`**.

Por ejemplo:

```
interface SumableRestableComparable extends Sumable, Comparable {
    SumableRestableComparable restar(SumableRestableComparable otro);
}

class Numero implements SumableRestableComparable {
    // ...

    @Override
    public Numero restar(SumableRestableComparable otro) {
        Numero otroNumero = (Numero) otro;
        return new Numero(num - otroNumero.num);
    }

    @Override
    public Numero sumar(Sumable otro) {
        Numero otroNumero = (Numero) otro;
        return new Numero(num + otroNumero.num);
    }

    @Override
    public boolean menorQue(Comparable otro) {
        Numero otroNumero = (Numero) otro;
        return num < otroNumero.num;
    }
}
```

1.6. Métodos predeterminados

1.7. Ejemplo: Interfaz `CharSequence`

1.8. Ejemplo: Clonación de objetos

1.8.1. `Cloneable`

1.8.2. `Object.clone()`

1.8.3. Constructor de copia

2. Paquetes y módulos

Bibliografía

Gosling, James, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. 2014. *The Java® Language Specification*. Java SE 8 edition. Upper Saddle River, NJ: Addison-Wesley.