Programación modular (II)

Ricardo Pérez López

IES Doñana, curso 2020/2021

Generado el 5 de mayo de 2021 a las 23:09:00

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.

La idea general de la programación orientada a objetos, y uno de sus principios fundamentales, es la abstracción.

Significa que los objetos del mundo real se pueden representar mediante sus modelos abstractos.

Diseñar modelos consiste en centrarse en las características esenciales de los objetos y descartar los demás.

Para entender lo que significa, echemos un vistazo a un lápiz.

Un lápiz es un objeto que podemos usar para dibujar. Otras propiedades como el material o la longitud pueden ser importantes para nosotros en ocasiones, pero no definen la idea de un lápiz.

Supongamos que necesitamos crear un programa de edición gráfica.

Una de las funciones básicas del programa es dibujar.

Antes de dibujar, el programa le pide al usuario que seleccione una herramienta de dibujo.

Puede ser un bolígrafo, lápiz, pincel, resaltador, marcador, aerosol y otros.

Cada herramienta de un conjunto tiene sus propias características específicas: un lápiz y un spray dejan marcas diferentes y eso importa. Pero también hay una característica esencial que los une: la capacidad de dibujar.

Ahora consideremos la clase `Lapiz`, que es una abstracción de un lápiz.

Como ya comentamos, la clase al menos debería tener un método `dibujar` que acepte un modelo de curva.

Esta es una función crucial de un lápiz para nuestro programa.

Supongamos que `Curva` es una clase que representa alguna curva:

```
class Lapiz {
    ...
    public void dibujar(Curva curva) {...}
}
```

Definamos clases para otras herramientas, por ejemplo, un pincel:

```
class Pincel {
    ...
    public void dibujar(Curva curva) {...}
}
```

Cada uno tiene método `dibujar`, aunque lo hace a su manera.

La capacidad de dibujar es una característica común a todos ellos.

Llamemos a esta característica `HerramientaQueDibuja`. Entonces podemos decir que si una clase tiene la característica `HerramientaQueDibuja`, entonces debería poder dibujar, lo que significa que la clase debería tener un método `void dibujar(Curva curva) {...}`.

Java permite declarar esta característica mediante la introducción de interfaces.

1.2. Definición de interfaces

En Java, una **interfaz** es un tipo referencia, similar a una clase abstracta, que sólo puede contener:

- Constantes estáticas públicas.
- Métodos abstractos públicos (sin implementación).
- Métodos predeterminados públicos (con implementación).
- Métodos estáticos públicos (con implementación).
- Métodos concretos privados (con implementación).
- Tipos anidados (clases anidadas, otras interfaces...).

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⟩ | ⟨tipo_anidado_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⟩ ::= ⟨método_predeterminado_público_interfaz⟩
                                | ⟨método_estático_público_interfaz⟩
                                | ⟨método_concreto_privado_interfaz⟩
⟨método_predeterminado_público_interfaz⟩ ::= [public] default ⟨def_método⟩
⟨método_estático_público_interfaz⟩ ::= [public] static ⟨def_método⟩
⟨método_concreto_privado_interfaz⟩ ::= private [static] ⟨def_método⟩
⟨tipo_anidado_interfaz⟩ ::= ⟨clase⟩ | ⟨interfaz⟩
```

Las **interfaces** sólo pueden tener dos **visibilidades**:

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

Los **miembros de una interfaz** sólo pueden tener dos visibilidades:

- **Pública**: es la visibilidad predeterminada salvo que se indique lo contrario, por lo que, si no se usa ningún modificador de visibilidad, se entiende que es `public`.
- **Privada**: sólo los métodos concretos pueden tener visibilidad privada.

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

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

Un **método abstracto** dentro de una interfaz no tiene cuerpo: es una declaración que acaba en `;` y no necesita el modificador `abstract`.

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

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

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

Los **métodos privados** tienen cuerpo, llevan el modificador `private`, pueden ser estáticos o de instancia, y se usan para descomponer otros métodos predeterminados o estáticos de la interfaz.

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, Comparable {
    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 tanto:

- Una clase puede implementar varias interfaces:

```
interface A { }
interface B { }
interface C { }

class D implements A, B, C { }
```

- Una interfaz puede extender una o varias interfaces usando **`extends`**:

```
interface A { }
interface B { }
interface C { }
```

```
interface E extends A, B, C { }
```

- Una clase puede extender otra clase e implementar varias interfaces, todo al mismo tiempo:

```
class A { }

interface B { }
interface C { }

class D extends A implements B, C { }
```

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

Los métodos predeterminados son un mecanismo de reutilización de código entre clases que no tienen por qué pertenecer a la misma jerarquía de generalización.

Cuando una clase implementa una interfaz con un método predeterminado, la clase «hereda» el método junto con su implementación.

Cuando una interfaz extiende a otra interfaz que contiene un método predeterminado, se puede hacer lo siguiente:

- No mencionar el método predeterminado, lo que hará que la subinterfaz herede el método y su implementación.
- Redeclarar el método predeterminado, lo que lo convertirá en un método abstracto.
- Redefinir el método predeterminado.

En el caso de que se implementen varias interfaces y ambas contengan métodos predeterminados con la misma signatura, la clase implementadora tendrá que especificar explícitamente qué método predeterminado se usará, o redefinir el método predeterminado.

```
interface TestInterface1
{
    default void show()
    {
        System.out.println("Default TestInterface1");
    }
}

interface TestInterface2
{
    default void show()
    {
        System.out.println("Default TestInterface2");
    }
}
```

```
class TestClass implements TestInterface1, TestInterface2
{
    public void show()
    {
        TestInterface1.super.show();

        TestInterface2.super.show();
    }

    public static void main(String args[])
    {
        TestClass d = new TestClass();
        d.show();
    }
}
```

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

1.9. Clases abstractas vs. interfaces

Las clases abstractas y las interfaces son herramientas para lograr la abstracción que nos permiten declarar métodos abstractos.

No podemos crear directamente instancias de clases abstractas ni de interfaces; sólo podemos hacerlo a través de clases que implementen los métodos abstractos.

Desde Java 8, una interfaz puede tener métodos predeterminados y estáticos que contienen una implementación, lo que hace que las interfaces sean más parecidas a clases abstractas.

Entonces, ¿cuál es la diferencia entre una interfaz y una clase abstracta?

Una **clase abstracta** puede tener métodos de instancia abstractos y no abstractos, mientras que una **interfaz** puede tener métodos de instancia abstractos o predeterminados.

Una **clase abstracta** puede extender una clase abstracta o concreta, pero una **interfaz** solo puede extender a otra interfaz

Una **clase abstracta** puede extender una sola clase, mientras que una **interfaz** puede extender cualquier número de interfaces.

Una **clase abstracta** puede tener variables finales, no finales, estáticas y no estáticas (de instancia), mientras que una **interfaz** sólo puede tener variables finales estáticas.

Una **clase abstracta** puede implementar una interfaz, pero no al revés (las **interfaces** no implementan).

Una **clase abstracta** puede tener un constructor, pero una **interfaz** no.

En una **clase abstracta**, la palabra clave `abstract` es obligatoria para declarar un método como abstracto, mientras que en una **interfaz** esta palabra clave es opcional.

La lista de diferencias anterior no es completa.

Las clases abstractas y las interfaces tienen muchas más diferencias, pero la principal es su **finalidad**:

- Normalmente, **las *interfaces* se usan** para desacoplar la interfaz de un módulo (la clase) de su implementación.

  Es decir: proporciona una interfaz estandarizada a los clientes de una clase, ocultando su implementación.

- En cambio, **las *clases abstractas* se usan** habitualmente como clases base que tienen miembros comunes para varias subclases.

Recuerda:

Una clase sólo puede extender a otra clase.

Una clase puede implementar una o varias interfaces.

Una interfaz puede extender a una o varias interfaces.

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.